Programarea calculatoarelor si limbaje de programare II

Module in Python

Universitatea Politehnica din Bucureşti

Sumar

- ***Avantajele utilizarii modulelor***
- Arhitectura programelor in Python
- Instructiunea *import*
- Directorul *__pycache__*
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Modulul in Python

- Reprezinta unitatea de program de nivelul cel mai ridicat
- Contine cod si date ce pot fi refolosite
- Ofera un spatiu de nume propriu, ceea ce ajuta la eliminarea conflictelor de nume ale variabilelor
- Orice fisier care contine cod Python este un modul
- Modulele pot importa alte module spre a accesa spatiul lor de nume
- Extensii codificate in alte limbaje, precum C, Java, C#, pot fi de asemenea module.
- Directoare, in cazul importului de pachete, pot fi si ele module

Modulul...

- Modulele sunt procesate cu:
  - Instructiunea ***import***, care permite importarea unui intreg modul
  - Instructiunea ***from***, care permite accesul la nume individuale dintr-un modul
  - Functia ***importlib.reload()***, care permite reincarcarea unui modul, fara a opri interpretorul de Python
  - Permit folosirea mai multor fisiere de cod pentru a forma un program
  - Toate numele globale dintr-un modul devin atribute ale obiectului de tip modul – la momentul importarii sale
- Avantajele folosirii modulelor:
  - Reutilizarea codului:
    - codul din fisierele modul este permanent, putand fi reincarcat si rulat de oricate ori

Modulul...

- modulele reprezinta un spatiu de nume, format din atribute ale modulului, ce pot fi referite de alte module externe – programare modulara
- Impartirea programelor in spatii de nume individuale:
  - numele din alte module nu sunt vizibile decat in urma importarii explicite
  - conflictul de nume este astfel evitat, fiindca chiar si in mod interactiv codul este inclus intr-un modul
- Modulele servesc la implementarea de servicii sau date partajate:
  - componentele partajate necesita o singura copie per program, e.g. o functie scrisa intr-un modul importat de alte module din program

Sumar

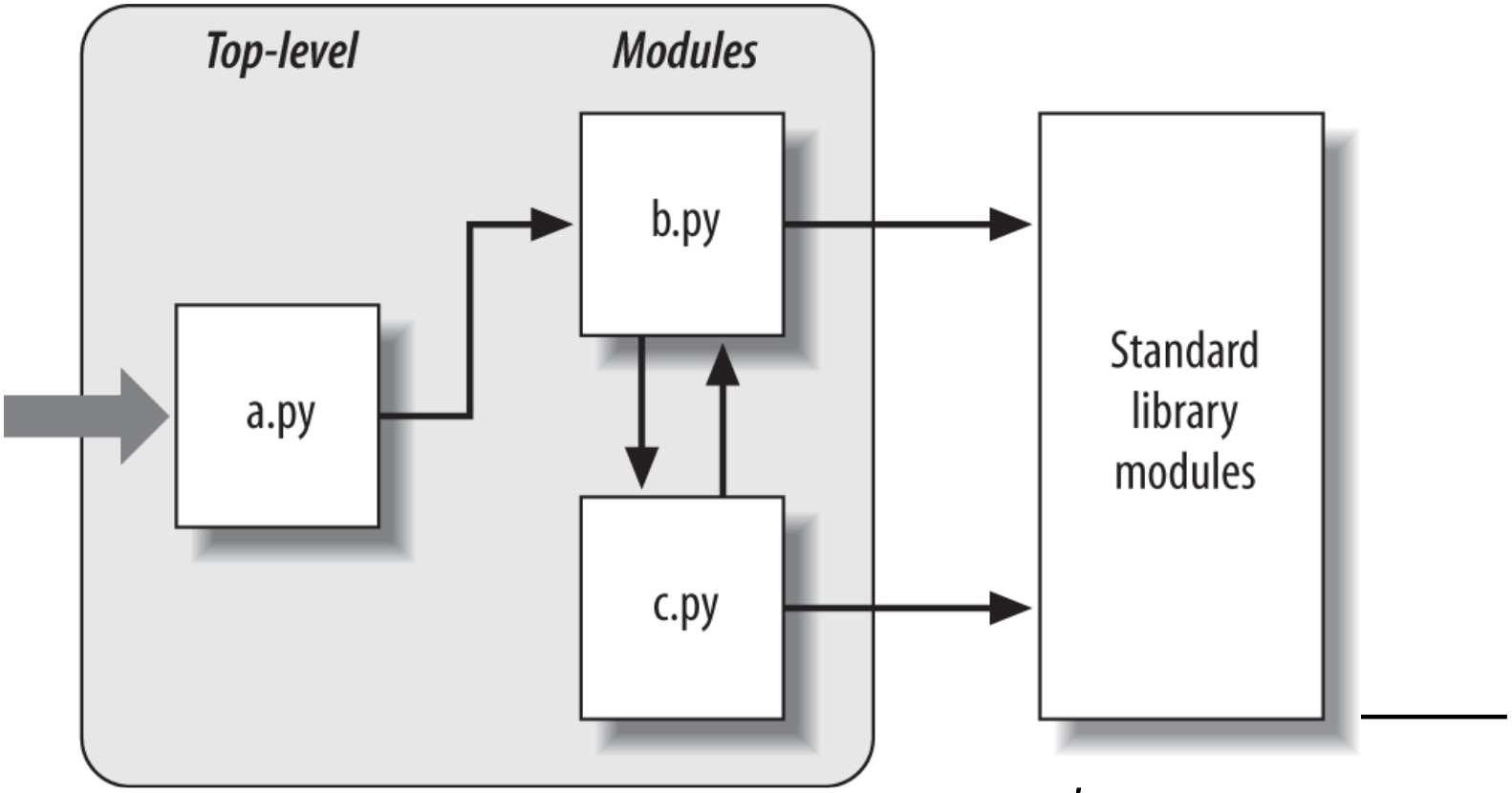
- Avantajele utilizarii modulelor
- ***Arhitectura programelor in Python***
- Instructiunea *import*
- Directorul *__pycache__*
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Structura programelor Python

- Un program Python este format din fisiere de cod ce contin instructiuni in limbajul Python, cu un fisier principal si zero sau mai multe fisiere modul
- Fisierul principal – scriptul Python, contine fluxul de control al programului, fiind folosit la lansarea in executie a programului
- Fisierele modul contin componente ale programului, folosite de scriptul principal sau/si alte module ale programului
- De obicei, fisierele modul nu executa nimic vizibil la importare, producand doar definitiile necesare in alta parte
- Importarile modulelor ne dau acces la atributele acestora

Importari si atribute ale modulului

- Fie **a.py** scriptul care se executa si care foloseste modulele **b.py** si **c.py**:



Importari...

 # Fisierul b.py

```
def spam(text):
    print(text, 'spam')
```

 # Fisierul a.py

```
import b
b.spam('gumby') # Afiseaza: gumby spam
```

- Efectul executiei instructiunii **import b** (din a.py):
  - Se incarca fisierul **b.py** (daca nu fusese deja importat) si accesul la toate atributele sale se va face cu variabila **b**, careia i s-a asignat obiectul modul rezultat in urma incarcarii
  - **b** identifica numele extern al fisierului – **b.py** (si alte extensii sunt posibile) si devine referinta catre obiectul modul incarcat
  - importarea consta in executarea tuturor instructiunilor din fisierul importat – precum si a altor fisiere cerute, la momentul executiei instructiunii *import*. Obiectele definite in modul sunt create tot la executie, iar numele globale devin atributele modulului, accesibile importatorului, e.g. *b.spam()*

Importari...

- Notatia *obiect.atribut* inseamna selectia/accesarea atributului desemnat cu operatorul **.** – pot fi atat functii cat si date
- Importarea poate fi generalizata: a.py importa b.py care importa c.py care poate importa b.py din nou, si asa mai departe, pe oricate nivele
- Modulele scrise pentru un program pot fi folosite si in alte programe, ceea ce inseamna ca putem avea un grad mare de reutilizare a codului (folositor in mai multe locuri)

Modulele bibliotecii standard

- Python ofera in mod automat un mare numar de module utile in dezvoltarea programelor, reprezentand biblioteca sa standard
- Peste 200 module, contin cod independent de platforma, e.g. interfata cu sistemul de operare, persistenta obiectelor, prelucrarea textelor, retea si scripting pentru Internet, programare GUI, etc.
- Se folosesc prin importarea modulelor necesare
- Documentatia bibliotecii – cu *help(), PyDoc, Idle* si on-line la http://www.python.org
- Informatii si in alte manuale de programare in Python, cautare cu Google, etc.

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- ***Instructiunea import***
- Directorul *__pycache__*
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Executia instructiunii *import*

- Instructiunea ***import*** este executabila (nu este insertie de text ca *#include* din C)
- Sunt trei pasi in executia unui *import*:
  1. **Gasirea** fisierului care contine modulul de importat
  2. **Compilarea** si obtinerea *byte code*-ului asociat – daca nu exista deja
  3. **Executia** codului, care construieste toate obiectele definite in modul
- Cei trei pasi se executa numai atunci cand un modul este importat pentru prima data intr-un program
- Obiecte modul existente se afla in tabela (dict) *sys.modules*, care este verificata spre a nu se relua cei trei pasi.

1. Gasirea fisierului

- Instructiunea *import* specifica doar numele fisierului, fara extensia *.py*, si fara calea completa de localizare (~~import c:\dir1\b.py~~) e.g. **import b**
- Python foloseste o *cale standard de cautare* a modulelor, *sys.path*
- Este incorecta introducerea caii de cautare sau a extensiei in import-ul clasic
  - importul pachetelor de module permite cai cu separator punctul
  - calea standard este folosita la alegerea punctului de plecare a caii
  - executabilele de tip *frozen* contin tot byte code-ul necesar in imaginea binara, deci cautarea modulelor nu conteaza

2. Compilarea

- Codul compilat se afla in fisiere cu extensia **.pyc**, care contin un numar magic (<= v3.1) sau numele compilatorului (>=v3.2) si au o data de creare, cu ajutorul carora se decide daca compilarea este necesara sau nu, prin comparatie cu data fisierului sursa si cu versiunea de Python curent executata
- **Compilarea**: daca sursa este mai noua sau daca versiunea de Python a modulului (.pyc) este diferita de cea executata curent. Fisierele cu extensia .pyc se afla intr-un subdirector numit __**pycache**__ din directorul sursei incepand cu v3.2, altfel chiar in directorul sursa
- **Nu se compileaza** daca sursa este mai veche si daca versiunea de Python corespunde cu cea rulata

2. Compilarea...

- Daca sursa lipseste, dar byte code-ul exista (in fisierul cu numele corespunzator si extensia .pyc), atunci acesta este incarcat direct. Astfel, un program poate fi livrat doar in byte code, fara surse.
- De notat ca fisiere cu extensia .pyc apar numai in urma unui import; scriptul principal este compilat in memorie si apoi byte codul este pierdut la incheierea programului
- Fisiere cu extensia .pyc actuale ajuta la sporirea vitezei de executie/lansare a unui program
- Si scriptul principal poate fi importat spre executare, rezultand un fisier cu extensia .pyc (vezi variabila __*name*__ si numele __*main*__ cand NU este importat)

3. Executia

- Este operatia finala a unui import, care executa byte code-ul modulului, in intregime, de la prima la ultima instructiune, rezultand un obiect de tip modul.
- Asignarile genereaza atribute pentru obiectul modul:
  - instructiunile *def* creeaza functii cu numele atribut al obiectului modul, putand fi apelate de importator
  - Codul “real” dintr-un modul poate produce rezultate vizibile la importare – *def* NU.
- Deoarece cei trei pasi sunt laboriosi, importul are loc o singura data per proces.
- Reincarcarea (in urma modificarilor sursei) se poate face numai cu functia ***importlib.reload()***

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- ***Directorul __pycache__***
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

__pycache__ in v3.2+

- Daca programul nu poate scrie pe disc, compilarea codului se face in memorie, iar executia este astfel posibila
- In v3.1 si mai vechi, byte code-ul se salveaza in acelasi director ca sursa, e.g. *module.pyc*
  - Fisierul include un camp “magic” care determina versiunea
  - Recompilarea se face daca versiunea de Python rulata difera, chiar daca sursa nu a fost modificata
- In v3.2 si mai noi, byte-code-ul se salveaza in subdirectorul ***__pycache__*** (creat automat daca lipseste) din directorul sursa, e.g. *__pycache__/module.cpython-37.pyc*
  - Deoarece versiunea de compilator este inclusa in nume, recompilarea se face cand se ruleaza o alta versiune, care lipseste, sau/si sursa este mai noua
  - Alte compilatoare: Jython, PyPY – *__pycache__/module.pypy36.pyc*

Exemple de *byte code*

- script0.py:

```
C:\>dir script0.py
 08/05/2019  11:42 AM                39 script0.py
C:\>rem Python 2.7:
C:\>py -2
>>> import script0
hello world
126765060022822940149670320537 6
C:\>rem script0.pyc alaturi de sursa script0.py:
C:\>dir script0*
08/05/2019  11:42 AM                39 script0.py
02/04/2020  05:12 PM               154 script0.pyc
```

```
C:\>rem Python 3.7:
C:\>py -3
>>> import script0
hello world
126765060022822940149670320537 6
C:\>dir script0.*
08/05/2019  11:42 AM                39 script0.py
02/04/2020  05:12 PM               154 script0.pyc
C:\>rem script0.cpython-37.pyc in subdirectorul
    __pycache__:
C:\>dir __pycache__\script0*
02/04/2020  05:21 PM               172
    script0.cpython-37.pyc
```

Exemple...

- Alt compilator, alt fisier byte code:

C:\>*rem PyPy 3.6*

C:\>**pypy3**

>>>> **import script0**

hello world

126765060022822940149670320537 6

C:\>*rem script0.pypy36.pyc in subdirectorul __pycache__:*

C:\>**dir __pycache__\script0.***

02/04/2020 05:21 PM 172 script0.cpython-37.pyc

02/04/2020 05:30 PM 179 script0.pypy36.pyc

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- Directorul __pycache__
- ***Cautarea modulelor***
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Calea de cautare a modulelor

- Este compusa din urmatoarele componente, concatenate, in ordine, unele configurabile:
  - Directorul curent al programului (auto)
  - Variabila de environment **PYTHONPATH** (daca setata) (config)
  - Directoarele bibliotecii standard de Python (auto)
  - Continutul fisierelor (text) cu extensia **.pth** (daca exista) (config)
  - Directorul **site-packages** (din distributia de Python, subdirectorul **Lib\site-packages**) (auto)
- Calea este cuprinsa in lista **sys.path**
- **Directorul curent** este initial cel care contine scriptul, daca executat ca program, sau chiar directorul curent in modul interactiv
  - Este cautat primul; poate masca biblioteca

Calea...

- Directoarele din **PYTHONPATH**, enumerate cu **;** (punct si virgula) intre ele
  - Este configurabila si de interes cand programele importa module din alte directoare
- **Directoarele standard** de Python, sunt intotdeauna cautate
- **Fisierele** text cu extensia **.pth**, contin cai de cautare, cate una per linie
  - Pot fi plasate in directorul *site-packages* sau in radacina distributiei (e.g. C:\Program Files\Python37)
  - Caile se pun in lista inainte de *site-packages*, dupa directoarele standard

Calea...

- Directorul **LIB\site-packages**, in care se instaleaza extensiile unor terte parti, ultimul cautat, automat.

Configurarea caii

- **PYTHONPATH** poate fi configurat la nivel de sistem, din Control Panel, sub Windows; dar si local (e.g. sub Linux, in scriptul **.bashrc**, **export PYTHONPATH=...*:*...**)
- Un fisier *C:\Program Files\Python37\pydirs**.pth*** poate contine directoare per fiecare linie
- Directorul curent este intotdeauna la inceputul caii de cautare

Lista **sys.path**

- Continutul listei ***sys.path*** este calea de cautare efectiva, la un moment dat, pentru gasirea modulelor de importat:

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files\\Python37\\Lib\\idlelib',
    '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']
```

- Modificand la executie lista *sys.path* (cu *sys.path.append()*, *sys.path.insert()*, etc) este posibila configurarea sa in mod dinamic:
  - De exemplu, serverele de WEB, limiteaza accesul user-ului *nobody* prin alterarea listei sys.path

Selectarea fisierelor modul

- Fiindca extensia modulului este omisa, se selecteaza orice tip de fisier cu numele din import, e.g. **import b**:
  - Fisierul sursa, **b.py**
  - Fisierul **b.pyc**, continand byte code
  - Fisierul **b.pyo**, contine cod optimizat (python -O ...)
  - **Directorul b**, in cazul importarii pachetelor
  - Un modul de tip extensie compilata in C, C++, etc, configurata dinamic, e.g **b.so** in Linux, **b.dll** sau **b.pyd** in Windows sau Cygwin
  - Un modul predefinit in C si configurat static
  - Un fisier ZIP, extras automat la importare
  - O imagine de memorie, pentru executabile *frozen*
  - O clasa Java, pentru Jython
  - O componenta .NET pentru IronPython

Selectarea...

- Au prioritate fisierele aflate mai intai pe cale, iar daca atat **b.so** cat si **b.py** sunt in acelasi loc, se aplica o regula generala de selectie, care poate varia cu versiunea de Python
  - Se recomanda folosirea de nume de modul distincte sau configurarea caii in mod explicit
- Fisiere cu extensia **.zip** pot fi importate, reprezentand o pozitie predefinita – *hook* – cu continut ajustabil – vezi *C:\\Program Files\\Python37\\python37.zip* din calea standard

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- Directorul __pycache__
- Cautarea modulelor
- ***Crearea modulelor***
- Utilizarea modulelor
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Cum se creeaza un modul

- Modulele sunt fisiere ce contin cod Python, editate cu un editor de text (NU de documente), e.g. *notepad*, si avand extensia **.py**
- Sunt usor de folosit, prin importarea intregului modul sau a unor nume din modul, si folosirea obiectelor referite

```
# Fisierul module1.py, creat cu notepad
def printer(x): # Atributul printer, o functie
    print(x)
```

- Numele modulelor trebuie sa fie variabile valide in Python
  - Nu cuvinte cheie (**nu** *if.py*)
  - Doar litere, cifre si underscore _

Cum...

- Nu spatii albe in numele modulului
- Numele extern complet al fisierului incepe cu un director din calea de cautare a modulelor, apoi numele din *import* si in final extensia *.py*

- Alte tipuri de module:
  - **Extensiile de module**, sunt scrise in alte limbaje, e.g. C, C++, Java, si contin biblioteci externe
    - Au acelasi comportament ca modulele din Python: se importa, se acceseaza functii si obiecte ca atribute ale modulului

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- Directorul __pycache__
- Cautarea modulelor
- Crearea modulelor
- ***Utilizarea modulelor***
- Spatiul de nume al modulelor
- Reincarcarea modulelor

Exemple

- Modulele se folosesc cu instructiunile **import** sau **from**
  - Ambele gasesc, compileaza si executa codul modulului, daca nu a fost deja incarcat
  - *import* acceseaza intreg modulul, iar numele din modul trebuie sa fie calificate (cu numele modulului)
  - *from* copiaza anumite nume – NU obiecte – dintr-un modul
- Instructiunea ***import m1, m2, ...*** (separate cu virgula):

```
>>> import module1 # Importarea intregului modul
>>> module1.printer('Hello world!') # Calificare cu numele modulului
Hello world!
```

- module1 a devenit variabila ce refera un obiect de tip modul
- Calificarea: **module1.printer** – atribut

Exemple...

- Instructiunea ***from module import a1, a2, ...*** (separate cu virgula):

```
>>> from module1 import printer # Copiaza atributul printer
>>> printer('Hello world!') # Fara calificare, direct apel de printer()
Hello world!
```

- Fara calificare inseamna scriere mai putina
- *from* face acelasi lucru ca *import*, dar adauga copierea unor nume din modul, pentru acces direct (dar numele modulului nu mai e disponibil)
- Instructiunea ***from module import ****:

```
>>> from module1 import * # Se copiaza TOATE variabilele din modul
>>> printer('Hello world!')
Hello world!
```

Exemple...

- In Python v3.x, **import *** se poate folosi numai pe nivelul cel mai de sus al unui modul, si nu intr-o functie, deoarece detectarea statica a variabilelor o face imposibila acolo.
- Este recomandabil ca toate importarile sa fie scrise la inceputul unui modul

import executa codul o singura data!

- Deoarece *import* si *from* sunt operatii costisitoare, codul modulelor este executat o singura data, la primul *import* sau *from*, per proces
- Importari ulterioare folosesc obiectul modul existent:

```
# Fisierul simple.py
print('hello')
spam = 1 # Initializare de variabila
```

```
C:\>python
>>> import simple # Primul import, incarca/executa codul modulului
hello
>>> simple.spam # spam este atribut al lui simple
1
>>> simple.spam = 2 # Modificare atribut din modul
>>> import simple # Rezulta acelasi obiect, deja incarcat
>>> simple.spam # Codul modulului NU mai este executat
2
```

import si *from* sunt asignari executabile

- Se comporta ca *def*, fiind instructiuni executabile:
  - se pot folosi in *if* (pentru selectii alternative), in *def* (pentru incarcare in timpul apelului), in *try* (pentru valori predefinite)
- Sunt atribuiri implicite (ca def):
  - *import* asigneaza unui nume obiectul modul
  - *from* asigneaza nume ce refera obiecte (cu acelasi nume) din modul
- Modificarea in-place a ob. modificabile din module:

```
# Fisierul small.py
x = 1
y = [1, 2]
```

```
C:\> python
>>> from small import x, y # Copierea numelor x si y
>>> x = 42 # Modificare a lui x, locala!
>>> y[0] = 42 # Modificarea in-place a lui y!
>>> import small # small refera ob. modul
>>> small.x # Este din modul, nu local
1
>>> small.y # Are alt continut acum!
[42, 2]
```

import...

- Modificarile variabilelor din alte module se poate face numai cu *import*, **nu** cu *from*, fiindca *from* produce nume locale (cu exceptia obiectelor modificabile schimbate in-place):

C>\>**python**

>>> **from small import x, y** *# Referire*

>>> **x = 42** *# x este local*

>>> **import small** *# small este numele modului*

>>> **small.x = 42** *# Modificarea atributului x din modulul small*

- Atentie, schimbarile directe din alte module sunt nerecomandabile

from si *import*

- *from* este echivalent cu urmatoarea secventa *import*:

```
from module import name1, name2 # Se copiaza numai name1 si name2
```

```
import module # Incarcare ob. modul
name1 = module.name1 # Copiere locala prin asignare
name2 = module.name2
del module # Stergere nume al modulului
```

- *from* executa incarcarea intregului modul in memorie, ca *import*, indiferent de cate nume copiaza
  - Performanta este practic neafectata, fiind vorba de byte code

Probleme cu instructiunea *from*

- *from* ascunde locatia reala a unei variabile
- Calificarea cu nume de module este mai anevoioasa e.g. cu **tkinter** – GUI, care are multe atribute
- *from* poate corupe spatiul de nume unde se executa, afectand variabilele cu acelasi nume
  - calificarea elimina aceste probleme
  - *from* cu listarea explicita a numelor importate este mai putin primejdioasa e.g. *from module import x, y, z*
- *from* are probleme cu *reload()* (vede in continuare aceleasi obiecte, chiar daca modulul a fost modificat si reincarcat)

Probleme...

- Recomandari:
  - se prefera *import* pentru modulele simple
  - se listeaza explicit variabilele din *from*
  - *import* * a se limita la un singur import per fisier – asa incat variabilele nedefinite vor face parte din acel *import* *
- *import* este obligatoriu (in loc de *from*) doar pentru a accesa acelasi nume din doua module diferite:

```
# Fisierul M.py
def func():
    ...executa Ceva...

# Fisierul N.py
def func():
    ...Executa Menelaus ...
```

```
# Fisierul O.py
from M import func
from N import func # Importul precedent este
    inlocuit
func() # Apel de N.func()!!
```

Probleme...

 # Fisierul O.py, corect!

import M, N *# Incarcare module*

M.func() *# Apeluri diferite acum*

N.func() *# Calificarea cu numele de modul le face diferite*

*# Fisierul O.py, cu clauza **as**, corect!*

from M import func as mfunc *# Redenumire unica cu "as"*

from N import func as nfunc

mfunc(); nfunc() *# Apeluri diferite! import cu as*

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- Directorul __pycache__
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- ***Spatiul de nume al modulelor***
- Reincarcarea modulelor

Fisierele genereaza spatii de nume

- Modulele corespund cu fisiere (cod Python)
- Numele din module reprezinta atributele acestora
- Asignarile globale din module – e.g. X = 1 in modulul M.py, produc atribute ale modulului, globale in modul – e.g. X si accesibile din afara modulului (dupa importare) – e.g. M.X
- Instructiunile dintr-un modul sunt executate cu ocazia primului import, cand se creeaza si obiectul de tip modul
- Atribuirile de nivelul cel mai de sus din modul creeaza atribute ale modulului (nu **din** *def* sau *class*) e.g. cu **=** sau cu ***def*** (de functie).

Fisierele...

- Spatiul de nume al unui modul se poate accesa cu atributul predefinit __**dict**__ sau cu functia **dir(M)**; dir afiseaza lista cheilor din dictionarul __dict__, dar este incompleta.
- Modulele reprezinta un spatiu global de nume, conform cu regula conturului LEGB, dar fara L si E (L este global). Spatiul de nume al modulului este permanent dupa import (spre deosebire de functii, unde spatiul de nume asociat exista doar pe durata apelului).

Fisierele...

- Exemple:

# Fisierul module2.py	**def func(): pass**
print('starting to load...')	**class klass: pass**
import sys	**print('done loading.')**
name = 42	
>>> **import module2**	42
starting to load...	**>>> module2.func**
done loading.	<function func at 0x00000242E8A1A288>
>>> module2.sys	>>> **module2.klass**
<module 'sys' (built-in)>	<class 'module2.klass'>
>>> module2.name	

- *sys*, *name*, *func* si *klass* au devenit atribute ale lui *module2* dupa import.

Dictionarul unui spatiu de nume de modul

- Spatiul de nume al unui modul este memorat cu un obiect de tip dictionar, accesibil cu **atributul** modulului numit **__dict__**, ale carui chei sunt numele din spatiul de nume al modulului:

```
>>> list(module2.__dict__.keys())
['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '__cached__',
    '__builtins__', 'sys', 'name', 'func', 'klass']
```

- '**__file__**' este cheia a carei valoare este numele complet al fisierului care contine modulul:

```
>>> module2.__dict__['__file__']
'C:\\Users\\Dan\\Desktop\\code\\module2.py'
```

- '**__name__**' este cheia a carei valoare este numele modulului:

```
>>> module2.__dict__['__name__']
'module2'
```

Dictionarul...

- Numele definite in modul se obtin filtrand numele delimitate de __:

```
>>> list(name for name in module2.__dict__.keys() if not name.startswith('__'))
['func', 'klass', 'name', 'sys']
>>> list(name for name in module2.__dict__ if not name.startswith('__')) # Expresii generator
['func', 'sys', 'name', 'klass']
```

- Atributele se pot accesa si prin indexarea dictionarului:

```
>>> module2.name, module2.__dict__['name']
(42, 42)
```

Sintaxa selectiei atributelor

- Accesarea oricarui atribut al unui obiect se face prin selectie/calificare: ***obiect.atribut***
- Regula conturului LEGB **nu** se aplica selectiei de atribute, ci doar numelui de obiect, fara calificare
- Reguli sintactice:
  - *Variabilele simple*, **X** – se cauta cu regula conturului LEGB
  - *Selectia/calificarea*, **X.Y** – se cauta X cu regula LEGB, apoi atributul Y este cautat in obiectul X
  - *Selectie/calificare multipla*, **X.Y.Z** – se evalueaza de la stanga la dreapta, Z este atributul obiectului X.Y, iar Y este atribut al lui X
  - *In general*, selectia/calificarea se aplica oricaror obiecte cu atribute: module, clase, extensii in C, etc.

Spatiu de nume vs. *import*

- Un fisier importat **nu** poate vedea numele din fisierul importator
  - Functiile **nu** pot vedea nume din alte functii, exceptand cazul functiilor definite in interiorul altor functii
  - Modulele **nu** pot vedea nume din alte module, exceptand cazul importarii modulelor

```
# Fisierul moda.py
X = 88 # X din moda
def f():
    global X # X din moda
    X = 99 # Nume externe invizibile!
```

```
# Fisierul modb.py
X = 11 # X din modb
import moda # Acces la numele din moda
moda.f() # Seteaza moda.X, nu modb.X!!
print(X, moda.X)
```

```
C:\>python modb.py
11 99
```

Spatii de nume incluse

- Un modul *mod1* care importa *mod2* si care la randul sau importa pe *mod3* este capabil sa vada variabila *mod2.mod3.X*:

 # Fisierul mod3.py:

X = 3

 # Fisierul mod2.py:

X = 2

import mod3 # Asignare obiect modul cu numele mod3

print(X, end=' ') *# X = 2*

print(mod3.X) *# X din mod3 – 3*

 # Fisierul mod1.py

X = 1

import mod2

print(X, end=' ') *# X = 1*

print(mod2.X, end=' ') *# X din mod2 – 2*

print(mod2.mod3.X) *# X din mod2.mod3 – 3*

Obs.: import mod2.mod3 nu se poate!

C:\>**python mod1.py**

2 3

1 2 3

Sumar

- Avantajele utilizarii modulelor
- Arhitectura programelor in Python
- Instructiunea import
- Directorul __pycache__
- Cautarea modulelor
- Crearea modulelor
- Utilizarea modulelor
- Spatiul de nume al modulelor
- ***Reincarcarea modulelor***

Reincarcarea modulelor

- Se poate face cu functia **importlib.reload()**:
  - *import* si *from* incarca/executa codul unui modul o singura data, la importare
  - Importari ulterioare refolosesc obiectul modul existent, fara reincarcari sau reexecutii
  - Functia *reload()* forteaza reincarcarea unui modul deja importat, care este modificat in-place cu noile asignari din codul modificat al modulului
- Permite modificari ale codului in regim dinamic, e.g. un program conectat la o baza de date care este modificat – depanat, fara a se relua initializarile
- Python poate reincarca modulele scrise in Python, insa extensiile (scrise in C) pot fi doar incarcate, nu si reincarcate

Sintaxa functiei *reload()*

- reload() este o functie, nu o instructiune
- *reload()* primeste ca argument un obiect modul existent
- *reload()* trebuie mai intai importata din modulul *importlib*

import module *# Importare initiala*

...folosirea modulului...

... # Fisierul module.py este modificat

...

from importlib import reload *# Importul functiei reload*

reload(module) *# Reincarcare/actualizare modul*

...folosirea atributelor modulului...

Sintaxa...

- La reincarcare, obiectul modul este modificat in-place, **nu** recreat. Alte referinte catre obiectul modul raman valabile:
  - reload() executa noul cod in spatiul de nume curent al obiectului modul – care nu este sters si recreat.
  - Noile asignari inlocuiesc nume (eventual existente) cu noi valori
  - Alte importari existente vor constata ca au aparut valori noi, dupa reload()
  - Importarile anterioare cu *from* raman nemodificate! Doar cele viitoare vor fi noi
  - reload() are un singur argument, deci reincarcarea altor module se face cu apeluri successive de reload()

Exemplu cu *reload()*

- Modificarea si reincarcarea modulului *changer.py* fara oprirea sesiunii Python interactive:

 # Fisierul changer.py initial:

```
message = "First version"
def printer():
    print(message)
```

```
>>> import changer
>>> changer.printer()
First version
```

 # Fisierul changer.py editat cu notepad:

```
message = "After editing"
def printer():
    print('reloaded:', message)
```

>>> # Sesiunea continua:

```
>>> import changer
>>> changer.printer() # Alt import nu are efect
First version
>>> from importlib import reload
>>> reload( changer )  # Returneaza obiectul de
    tip modul, ignorat...
<module 'changer' from
    'C:\\Users\\Dan\\Desktop\\code\\changer.
    py'>
>>> changer.printer()
reloaded: After editing
```

Utilizari ale *reload()*

- Reincarcarea modulelor, atunci cand repornirea unei aplicatii este costisitoare, e.g. servere de jocuri on-line
- Aplicatii GUI, e.g. modificare de widget (callback) cu GUI ramas activ
- Python este un limbaj pentru adaptari ale unor sisteme mari, fara recompilare sau fara sursa, si fara oprire si restartare de sistem.