Introducere în limbajul Python

Obiective

Scopul primelor trei laboratoare de Arhitectura Sistemelor de Calcul îl reprezintă familiarizarea cu probleme de multi-threading, concurență și sincronizare. Pentru a aplica mai ușor și mai rapid aceste noțiuni, vom folosi limbajul Python. În cadrul primului laborator vom prezenta principalele caracteristici ale limbajului Python, urmând ca în laboratoarele 2 și 3 și la tema 1 să lucrăm cu thread-uri pentru a implementa aplicații concurente.

Ce este Python?

Python este un limbaj de programare foarte popular (statistici github, utilizare), oferind posibilitatea programării structurate dar și orientate pe obiect și incluzând și elemente din paradigma funcțională. Este un limbaj de scripting, ceea ce înseamnă că este interpretat și nu compilat, economisind mult timp în procesul de dezvoltare și debugging.

Limbajul combină o putere remarcabilă cu o sintaxă foarte clară. Are module, clase, excepții, tipuri dinamice și garbage collection. API-ul Python oferă module pentru o gamă foarte mare de funcționalități, de la cele de bază pentru lucrul cu șiruri și fișiere, până la cele pentru lucrul cu procese, threaduri, sockets, serializare etc. Multe dintre aceste module oferă o interfață foarte similară programării la nivel de sistem din C (ceea ce se studiază la SO), astfel încât funcțiile Python au același nume si aproximativ aceiași parametrii ca cei pentru funcțiile din C pentru apeluri de sistem. Există o varietate de API-uri, cele mai populare la momentul actual fiind cele pentru aplicatii web, AI bots, data science si grafice, de exemplu:
wit.ai, flask, NumPy, SciPy, pandas, plotly, Matplotlib, Python & RaspberryPi.

Implementarea principală a Python, CPython (în C) a fost concepută pentru a fi portabilă: funcționează pe Linux, Unix, Windows, Mac OS X. Programele pot fi executate, de asemenea, și pe mașina virtuală Java prin folosirea compilatorului Jython, care le transformă în bytecode Java și permite folosirea de clase Java în cadrul acestora. Similar, există IronPython pentru .NET.

De ce Python?

Python este un limbaj ce oferă foarte multe funcționalități și are o curbă de învățare rapidă atât pentru programatorii ce cunosc deja limbaje precum C și Java, cât și pentru începători. Deși este un limbaj interpretat, acest lucru nu a stat în calea popularității sale și folosirii în numeroase proiecte. Puteți utiliza Python atât pentru scripturi, dezvoltarea unor infrastructuri de testare, cât și pentru aplicații web și machine learning/data mining.

De ce Python la ASC?

În cadrul laboratoarelor dorim să ne axăm pe scrierea de programe multi-threaded corecte, nu pe

particularitățile vreunui limbaj. Din acest punct de vedere Python este de preferat datorită interfeței simple pentru lucrul cu thread-urile și pentru modul de lucru mai rapid si numărul de linii de cod mult mai mic în comparație cu Java sau C. Avantajul este cel al simplității și asemănarea API-ului pentru thread-uri cu cel din C-pthreads, însă are și un dezavantaj oferit de un mecanism din interpretor, pe care îl vom discuta în laboratorul 2.

Instalare

În cadrul laboratoarelor vom lucra doar cu **Python 2.6 sau 2.7**, nu și cu versiunile >= 3. Python *nu* oferă *backwards compatibility* astfel încât programe scrise pentru o versiune pot să genereze erori de interpretare pe o altă versiune (de exemplu tratarea excepțiilor în versiuni >= 2.6 are altă sintaxă decât în 2.4). De asemenea, Python3 este în mod intenționat incompatibil cu versiunile 2.x, explicații asupra acestui lucru puteți citi și în articolul lui Guido van Rossum, autorul Python-ului.
Python se poate descărca de pe site-ul oficial, secțiunea Download.

Pe un sistem Linux, este foarte probabil să fie deja instalat. Verificați acest lucru dintr-o consolă:

```
$ python
Python 2.6.6 (r266:84292, Jan 22 2014, 05:06:49)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Dacă nu îl aveți instalat, pasul următor ar fi să-l căutați în repository-urile distribuției cu care lucrați. Exemplele următoare prezintă instalarea pe o distribuție bazată pe Debian și pe una bazată pe Red Hat.

```
$ sudo apt-get install python
```

```
$ sudo yum install python
```

Pe un system Windows se poate descărca versiunea 2.7.11. După instalare se poate adăuga calea către executabilul Python în global PATH, pentru a putea fi găsit de oriunde.

Tools

Dacă doriți să folosiți un IDE pentru Python vă recomandăm versiunea community a PyCharm.

Pentru a verifica respectarea code-style-ului recomandat de Python vă recomandăm PEP8 online.
Pentru verificarea code-style-ului, detectarea erorilor și alte lucruri utile, puteți folosi Pylint. Aceste este instalat și pe calculatoarele din laborator, la fel ca și PyCharm.

Cum se execută un program Python

În Windows, dacă extensia `.py` este deja înregistrată, un dublu-click pe numele scriptului este suficient. Se poate edita programul folosind GUI-uri ca IDLE (Python GUI) și Eclipse sau editat în notepad++ și executat din consolă (Start→Run→cmd).

Dacă folosiți Linux, există mai multe posibilități:

- fie se lansează interpretorul cu numele scriptului:

```
$ cat hello.py
print 'Hello World!'
$ python hello.py
Hello World!
```

- fie prima linie din program este de forma #!/calea/catre/interpretor iar scriptul se face executabil:

```
$ cat hello.py
#!/usr/bin/python
print 'Hello World!'
$ chmod +x hello.py
$ ./hello.py
Hello World!
```

Pentru a testa anumite funcționalități sau a verifica o anumită sintaxă nici nu este nevoie să scrieți un script, puteți folosi direct **consola** Python. Din ea se iese scriind `quit()` sau apăsând `Ctrl-D`.

Din consolă puteți obține **documentația** despre anumite module, clase, metode folosind `help(nume)`, inclusiv despre modulele și metodele scrise de voi dacă includeți în cod comentarii docstring (similare javadoc-ului din Java). Alternativ, dacă doriți să vedeți metodele disponibile pentru un anumit obiect puteți folosi funcția `dir(obiect)`, ca în exemplele de aici.

```
>>> import io
>>> help(open)
```

Unul din avantajele Python este că puteți folosi consola atunci când vreți să testați o anumită funcționalitate sau când nu sunteți siguri ce se întâmplă când folosiți anumite funcții/construcții. De exemplu puteți încerca diverse metode de parsare string-uri înainte de a le încorpora în programul vostru, sau poate pur și simplu vă puneți întrebări de genul “Pot pune într-un dicționar chei de tipuri diferite, unele integer unele string?” –> creați un astfel de dicționar în consolă și observați dacă merge.

Pentru a folosi într-un program Python argumente date din linia de comandă trebuie inclus modulul sys și folosită lista `argv`, care conține toate argumentele, pe prima poziție fiind numele scriptului.

```
import sys
if len(sys.argv) < 2:
```

```
    print "Usage:...."
    exit(0)
```

Particularități de sintaxă

Indentarea

- Indentarea în Python este mai mult decât parte a stilului de programare, este chiar parte din sintaxă
- O linie nouă termină o declarație, pentru a continua o declarație pe mai multe linii, se foloseste caracterul "\".
- Un bloc de cod are toate liniile indentate cu același număr de spații (nu există `begin` și `end` sau `{}`). Instrucțiunile dintr-un bloc de cod vor fi grupate unele sub altele, pe același nivel de indentare.
- Definițiile neindentate într-un program Python vor fi de obicei variabile globale, definiții de clase, proceduri, funcții sau părți ale *main*-ului.
- După instrucțiuni nu este obligatoriu să puneți simbolul `';'` precum în alte limbaje de programare, indentarea corespunzatoare fiind suficientă.
- Este foarte important sa nu uitați simbolul `':'` care precede o indentare

Structurile de control, definirea de rutine, clase, toate necesită o mai mare grijă în scrierea codului. Exemplu de folosire a indentării pentru a delimita blocurile de cod:

```
a = 1
# Afisam numerele de la 1 la 10
while a <= 10:
    print a,
    a += 1
print "Am terminat"
```

Comentarii

- O singură linie se poate comenta în Python folosind simbolul #.
- Pentru comentarii multi-line se folosesc trei ghilimele succesive `"""` sau `'''` la inceputul și respectiv la sfârșitul zonei de comentat.
- Comentariile cu ghilimele succesive din interiorul unui bloc de cod trebuie să fie pe același nivel de indentare cu acesta, altfel veți avea `IndentationError` la execuție. Comentariile cu # nu au această restricție.

```
[...]
if b > 3:
""" Silly comment """
```

```
    a = b
```

```
$python comments.py
  File "comments.py", line 4
    """ Silly comment """
                        ^
IndentationError: expected an indented block
```

Corect este:

```
if b > 3:
    """ Silly comment """
    a = b
```

Code style

- Pe site-ul oficial găsiți recomandările de code style, scrise chiar de creatorul limbajului, Guido van Rossum.
- statistici cu folosirea diverselor convenții de stil în proiectele Python de pe GitHub.

Tipuri de date și variabile

Python oferă tipuri de date numerice, booleene, șiruri (string, liste etc), dicționare, fișiere, clase, instanțe și excepții.

Din punct de vedere al tipării Python folosește tipuri pentru obiecte, însă la definirea variabilelor nu trebuie precizat tipul acestora. Constrângerile de tip sunt verificate la execuție (***late binding***), astfel încât pot apărea erori și excepții generate de folosirea unui tip necorespunzător în atribuiri și excepții.

Python asociază numele unei variabile cu un obiect, care poate fi număr, șir de caractere sau ceva mai complex. Când este folosită o variabilă, tipul acesteia este tipul obiectului cu care este asociată. Este greșită folosirea într-o expresie a unei variabile care nu a fost asociată cu un obiect.

```
i = 5         # i va fi de tip Integer
i = "Hello"   # i va fi de tip String
```

Alocarea și dezalocarea de memorie se face automat de către Python, existând un mecanism de *garbage collection*.

În Python se poate face *atribuirea de valori unor mai multe variabile simultan*:

```
x, y = 2, 3
```

Datorită faptului că partea dreaptă a unei expresii este evaluată înainte de a se face atribuirea,

valorile a două variabile pot fi foarte ușor interschimbate, fară a avea nevoie de o a treia variabilă:

```
x, y = y, x
```

Python este un limbaj ***dynamically typed***, după cum am arătat și mai sus, dar și ***strongly typed***, nepermițând tipului unei variabile să se schimbe atunci când este folosită în operații, decât printr-o conversie explicită. De exemplu: `'12' + 10` va da excepție la execuție, trebuie folosit `int('12') + 10`.

Stringuri

Pentru lucrul cu șiruri de caractere, Python oferă tipul String și o serie de operații de bază pe acesta, descrise în documentație.

Stringurile sunt incluse între ghilimele `"` sau `'`. Stringurile ce conțin mai multe linii sunt înconjurate de trei ghilimele succesive `"""` sau `'''`.

```
s = "string pe o linie"
linie = """string
pe mai multe linii"""
```

Stringurile sunt tratate ca niște liste, `cuvant[i]` fiind caracterul din șir ce are indexul `i`, unde `i` ia valori în intervalul `[-length,length)`. Folosirea unei valori în afara acestui interval va genera o eroare: `IndexError: string index out of range`. Python permite și folosirea unor indici negativi, care indexează elementele pornind de la sfârșitul șirului, precum în exemplul de mai jos

```
stringul:         s   t   r   i   n   g
index:            0   1   2   3   4   5
index negativ:   -6  -5  -4  -3  -2  -1
```

Nu există un tip de date special pentru caractere, acestea sunt văzute drept șiruri de lungime 1. Lungimea unui șir se poate afla cu ajutorul funcției: `len(some_string)`.

Caracterul `:` specifică un **subșir** al unui şir folosind sintaxa: `some_string[x:y]`. Subșirul obținut conține toate caracterele din șirul inițial (`some_string`) între pozițiile `x` si `y-1` (inclusiv). Dacă nu se specifică x sau y, acestea au implicit valorile 0, respectiv lungimea șirului.

String-urile sunt *immutable*. O dată create ele nu mai pot fi modificate.

```
>>> s="abcdef"
>>> s[0]="g"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item assignment
```

Particularități de lucru cu șirurile de caractere:

- pot fi **concatenate** folosind simbolul **+**
- pot fi **multiplicate** folosind caracterul *
- se poate folosi operatorul **%** pentru formatarea șirurilor similar printf-ului din C
- modulul string oferă șiruri predefinite și metoda **`format`** pentru crearea șirurilor (utilă în special când vrem să incorporăm în șir mai multe variabile). Atât la folosirea acestei metode, cât și a operatorului % nu este nevoie de conversia explicită la string a variabilelor.
- modulul str oferă metode de lucru cu șirurile precum `index`, `replace`, `split`, `isdigit`, `format` (similar format-ului din modulul string)
- modulul re - regular expressions este util pentru căutarea sau separarea unor substringuri folosind expresii regulate.

lab1_string_example.py

```
s = "string"
print s[0:2]        # st
print s[:3]         # str
print s[3:]         # ing
s2 = "one"

print "Write " + 2*s2 + " " + s                 # Write oneone string
print "hello %s, lab %d  !" % ("students",1)    # hello students, lab 1
!

import string
print "hello {}, lab {}".format("world",1)      # hello world, lab 1
print "hello {0}, lab {1}".format("world",1)    # incepand cu python 2.6
```

Tipuri de date numerice

În Python se poate lucra cu numere întregi și numere în virgulă mobilă. Numerele întregi dintr-o expresie se convertesc automat în numere în virgulă mobilă dacă este necesar. Cele patru tipuri numerice *int*, *long*, *float* și *complex* și operațiile acestora sunt descrise în API. În afară de operațiile aritmetice standard (+,-,*,/), există operatori pentru modulo **%** și pentru ridicarea la putere ****.

Deoarece Python este *strongly typed*, nu se face conversie automată între tipurile de date dintr-o expresie, cum ar fi operațiile între tipurile String și Integer. Pentru acest lucru se folosesc funcțiile `int(some_string)` pentru transformarea din string în integer și respectiv: `str(some_int)` sau `repr(some_int)` sau `` `some_int` `` (` - backquote) pentru transformarea din integer în string. **Atenție!** În cazul transformării din string în integer, dacă stringul ce trebuie convertit conține și alte

caractere decât cifre, la execuție va apărea o eroare: `Value Error: invalid literal for int() with base 10`.

lab1_num_example

```
print "Un string:", "4" + "2"                  # Un string: 42
print "Un numar:", 4 % 3 + int("41")           # Un numar: 42
print "Un string:", "4" + str(2)               # Un string: 42
print 'persoana %s are %d ani' % ("X", 42)     # persoana X are 42 ani
print (2.0 + 3.0j) * (2.1  -6.0j)              # (22.2-5.7j)
print 2 ** 3 ** 2                              # 512
```

Tipurile de date numerice și șirurile de caractere sunt obiecte **immutable**, adică starea lor (valoarea) nu mai poate fi schimbată după ce sunt construite (ca și în cazul String și tipurilor numerice din Java). Mai multe detalii găsiți în Python datamodel. Pentru un exemplu de *mutable vs immutable* puteți urmări exemplul de la secțiunea Liste și tupluri.

Liste și Tupluri

Python pune la dispoziție două tipuri de structuri pentru a reprezenta liste de elemente: tupluri și liste. Diferența principală dintre cele două tipuri este că tuplul nu mai poate fi modificat după ce a fost declarat, fiind obiect immutable.

- lista se declară folosind paranteze drepte: `[]`.
- tuplul se declară folosind paranteze rotunde: `()`.
- **elementele** unei liste sau ale unui tuplu pot fi de **tipuri diferite**.
- nu există vectori în Python, însă există un modul *array* care se comportă ca o listă cu restricții asupra tipului elementelor din ea.
- *design tip*: de obicei tuplurile se folosesc pentru date heterogene, iar listele pentru date omogene (toate de același tip)

```
lista = ["string", 10]
tuplu = ("string", 10)
```

Accesarea elementelor unei liste sau unui tuplu se face la fel ca în cazul șirurilor de caractere, cu indecși pozitivi, negativi sau folosind operatorul `:`. Spre deosebire de stringuri și tupluri, elementele unei liste pot fi modificate cu ajutorul accesării prin indecși.

Operații utile (lista completă este dată în documentație):

- `len(lista)` - întoarce numărul de elemente din listă
- `lista.append(x)` - adaugă un element la sfârșitul listei
- `lista.extend(alta_lista)` - concatenează alta_lista la lista
- `lista.sort()` - sortează elementele listei
- `lista.reverse()` - inversează ordinea elementelor din listă
- afișarea unei liste se poate face folosind un simplu `print <nume_lista>`.

- putem referi o **sublistă** a unei liste, ajutându-ne de doi indecși: `sublista = lista[index1:index2]`

lab1_list_example

```
alta_lista = [1,2,3,4]
lista = []
lista.append(5)             # lista va fi [5]
lista.extend(alta_lista)    # lista va fi [5, 1 ,2, 3, 4]
del lista[0]                # lista va fi [1, 2, 3, 4]
del lista[0:2]              # lista va fi [3, 4]
print [1,2,3][2:2]          # afiseaza []
print [1,2,3][0:2]          # afiseaza [1, 2]
print [1,2,3][2:5]          # afiseaza [3]
```

Elementele unui tuplu nu mai pot fi modificate după ce au fost declarate. Din acest motiv nicio funcție prezentată mai sus ce modifică elementele unei liste nu poate fi aplicată asupra tuplurilor. Exemplul de mai jos ilustrează faptul că listele pot fi modificate și sunt *mutable*, iar tuplurile *immutable*.

```
>>> a = [1,2,3]
>>> b = a
>>> b[1] = 5
>>> b
[1, 5, 3]
>>> a
[1, 5, 3]
>>> t = (1,2,3)
>>> t[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

O nouă listă poate fi creată din altă listă folosind ***list comprehensions***: o expresie urmată de o clauza `for`, apoi de 0 sau mai multe clauze `for` sau `if`.

```
>>> print [ elem*2 for elem in [1,2,3,4] if elem!= 3]
[2, 4, 8]
>>> print [(x, x**2) for x in [1,2,3,4]]
[(1, 1), (2, 4), (3, 9), (4, 16)]
```

Dicționare

O structură dicționar este un set *neordonat* de *chei* și *valori* în care valoarea poate fi cautată

folosindu-se cheia. Cheile sunt obiecte hashable, ceea ce presupune că au o funcție hash ce întoarce întotdeauna aceeași valore pentru obiectul respectiv. **Atenție!** listele, dicționarele și alte tipuri de date *mutable nu* sunt *hashable*, deci folosirea lor drept chei nu este posibilă.

- dicționarele sunt declarate folosind acolade, elementele sunt de forma `cheie:valoare`, despărțite de virgule.
- ordinea de stocare este arbitrară, iar cheile pot să fie de tipuri diferite

```
dict={}
dict[0] = "primul"
dict["unu"] = 2
print dict           # {0: 'primul', 'unu': 2}
dict2 = dict
dict2[3] = "ceva"

del dict2["unu"]
print dict                  # {0: 'primul', 3: 'ceva'}
print len(dict)             # 2
print dict.has_key("5")     # False
if 0 in dict:
    dict["3"] = 2
print dict                  # {0: 'primul', '3': 2, 3: 'ceva'}
print dict.keys()           # [0, '3', 3]
```

Câteva operații utile (lista completă o găsiți în documentație):

- `len(some_dict)` - pentru a afla dimensiunea unui dicționar
- `del some_dict[cheie]` - pentru a șterge o intrare din dicționar
- operatorul `in` - pentru a afla dacă o cheie este în dicționar. Similar se poate folosi funcția `has_key(cheie)`
- `keys()` - întoarce o listă ce conține toate cheile din dicționar

Structuri condiționale

```
if conditie1:
    instructiuni
elif conditie2:
    instructiuni
else:
    instructiuni
```

- Pot exista mai multe secțiuni `elif` sau nici una, iar secțiunea `else` apare o dată sau niciodată.
- Numarul 0, listele și tuplurile goale, string-urile vide și valoarea None sunt considerate `False` dacă sunt folosite în evaluarea unei condiții.

Python nu oferă vreo construcție de tip *switch*, iar operatorul condițional nu are forma `?:`, ci `x if cond else y` .

```
>>> a = 1
>>> b=0 if a>0 else 2
>>> b
0
```

Structuri repetitive

Instrucțiunea "for"

Instrucțiunea `for` iterează după elementele unei secvențe, fie ea String, list sau tuple.

```
for elem in lista:
    instructiuni
```

În partea de după `in` din for-uri se folosesc și funcții care întorc liste, ca de exemplu `range()`:

- `range(stop)` sau `range(start, stop [,step])` - formează o listă cu elemente mai mici ca `stop` în progresie aritmetică, cu rația `step`. Primul element al listei este `start`. Implicit `start` este 0 si rația este 1.

```
for i in range(len(s)):
    print s[i]
```

Instrucțiunea "while"

Instrucțiunea `while` continuă iterația cât timp condiția specificată este adevarată.

```
while conditie:
    instructiuni
```

Instrucțiunea `break` termină forțat orice buclă `while` sau `for`, iar instrucțiunea `continue` sare la următoarea iterație.

Instrucțiunile `while` și `for` pot avea o clauză `else`. Aceasta se execută când se termină lista după care se face iterația `for` sau atunci când condiția `while` a devenit `False`. Instrucțiunile din clauza `else` nu se execută în cazul în care bucla este terminată printr-o instrucțiune `break`.

De ce avem nevoie de așa ceva? În exemplul următor avem cazul clasic în care ieșim din buclă când se îndeplinește o condiție și vrem să știm în codul de după buclă dacă s-a ajuns la acea condiție sau nu.

```
    s = [1,2,3,0,3,4]
```

```
    exists = False
    for i in s:
        if i == 0:
            exists = True
            break
    if exists:
        print "Exista"
    else: print "Nu exista"
```

Folosind clauza `else` nu mai avem nevoie nici de variabila intermediară, nici de verificarea acesteia după buclă.

```
    s = [1,2,3,0,3,4]
    for i in s:
        if i == 0:
            print "Exista"
            break
    else:
        print "Nu exista"
```

Funcții

Funcțiile sunt definite folosind cuvântul cheie `def` urmat de o listă de argumente și apoi de ":".

- lista de argumente conține doar numele acestora, tipul lor fiind cel trimis la apelul funcției.
- se pot specifica **valori implicite** pentru parametrii unei funcții, iar dacă la apelare nu se include acel parametru este luată valoarea implicită
- dacă nu se include `return` sau se execută `return` fară argumente, atunci funcția va întoarce `None`.
- dacă o metodă vreți să o lăsați fără implementare, folosiți keyword-ul `pass`

Funcțiile în Python sunt obiecte!. Le putem transmite ca argumente si atribui. Observati __class__ in exemplul de mai jos.
>>> def hello_asc(): print "hello"

>>> dir(hello_asc)

['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__doc__', '__format__', '__get__', '__getattribute__', '__globals__', '__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals', 'func_name']

>>> f = hello_asc

```
def fractie(x, y=1):   # y are valoare implicita
    if (y==0):
```

```
        return          # va întoarce None
    else:
        return float(x)/float( y)
def fractie2():
    #TODO
    pass

print fractie(6, 4)
print fractie(6)        # y va fi 1
```

Funcțiile mai pot fi apelate folosind mapări de forma: **nume_parametru = valoare**. Pentru exemplul de mai sus se mai poate apela funcția și astfel:

```
fractie(y=7, x=5)       # in cazul acesta nu mai conteaza ordinea
fractie(6, y=1)
```

Dacă o funcție are un parametru de forma `*lista`, atunci la apelare acesta va conține un **tuplu de argumente**, ca în exemplul următor:

```
def suma(*lista):
    s=0
    for i in lista:
        s=s+i
    return s

print suma(2,3,5)
```

Dacă o funcție are un parametru de forma `**nume`, atunci la apelare acesta va conține un **dicționar de argumente**, ca în exemplul următor:

```
def afisare(**nume):
    for i in nume.keys():
        print i,':',nume[i]

afisare(client="Alex", vanzator="Alina")
```

Execuția instrucțiunilor de mai sus va afișa:

```
client : Alex
vanzator : Alina
```

Python oferă o serie de funcții predefinite (built-in functions) cum ar fi: `len` pentru lungimea unui obiect de tip colecție, `id` care în implementarea CPython întoarce adresa de memorie a obiectului, funcții de conversie de tip cum ar fi `str`, `int`, `float`, `bool`, etc.

Variabile globale

Pentru a accesa variabilele globale ale programului într-o funcție, trebuie să folosim cuvântul cheie **`global`** cu sintaxa: `global some_var`.

Nu este necesară folosirea acestui cuvânt cheie atunci când în funcție doar se citește variabila respectivă. Este obligatorie folosirea `global` doar dacă în funcție se dorește modificarea valorii variabilei. Fără această declarație s-ar crea o nouă variabilă locală funcției, care nu ar afecta valoarea variabilei globale.

Includerea de cod extern

În Python se pot include funcții, variabile și clase definite în alte fișiere. Un astfel de fișier, ce poate fi inclus, poartă denumirea de modul. Exemplu:

```
import random
```

Atunci când includem modulul putem să îl folosim cu un alt nume, eventual, pentru a evita coliziuni de denumiri:

```
import random as rand
```

Instrucțiunea `import` din exemplul de mai sus nu încarcă în tabela de simboluri numele funcțiilor definite în modulul `random`, ci doar numele modulului. Folosind acest nume se pot accesa funcții definite în interiorul modulului folosindu-se sintaxa `nume_modul.nume_functie(parametri)`.

```
random.random()
```

Un alt mod de a include este folosind `from <nume_modul> import <lista_constructii>`, în felul acesta vom avea acces doar la clasele/metodele din modul precizate în <lista_constructii>. Această modalitate este indicată atunci când un modul are un număr foarte mare de clase. Un alt avantaj al acestei metode este că nu se mai folosește numele modulului atunci când se utilizează această construcție, ca în exemplul de mai jos:

```
from random import random # se include doar metoda random()
random()                  # se apeleaza metoda random
```

Excepții

Ca și Java și alte limbaje de nivel înalt, Python oferă suport pentru excepții. Pentru prinderea excepțiilor aruncate de către funcții se folosește mecanismul `try-except`, asemănator celui `try-catch` din Java.

```
try:
```

```
    x = int(buffer)
except ValueError as e:
    print "Date de intrare invalide"
finally:
    # clean-up
```

Mecanismul funcționează în felul următor:

- se execută instrucțiunile din blocul `try`
- dacă apare o excepție tratată de un bloc `except`, execuția sare la instrucțiunile din blocul respectiv. După ce excepția este tratată, execuția continuă cu prima instructiune de după blocul `try`
- dacă apare o excepție ce nu este tratată de niciun bloc `except`, aceasta continuă să fie propagată.
- instrucțiunile din blocul `finally` se execută întotdeauna, indiferent dacă a fost prinsă o excepție sau nu, iar excepțiile netratate se aruncă dupa ce se execută blocul `finally`. Blocul acesta este opțional și este folosit pentru acțiuni de “clean-up”, de exemplu închiderea fișierelor.

O excepție poate fi aruncată folosind instrucțiunea `raise`. Aceasta poate fi folosită și fară argumente în interiorul unui bloc `except` pentru a re-arunca excepția prinsă de blocul respectiv.

```
if (j>100):
    raise ValueError(j)
```

O instrucțiune `try` poate avea mai multe clauze `except`. Ultima clauză `except` poate să nu aibă specificată nicio excepție de tratat fiind astfel folosită pentru a trata toate exceptiile netratate de celelalte clauze.

Pe lângă `while` și `for`, și construcțiile `try-except` pot avea opțional și o clauză `else`. Instrucțiunile din blocul `else` sunt executate atunci când blocul `try` nu generează nicio excepție.

Python oferă un set de excepții predefinite (*built-in*), conținute în **modulul `exceptions`** (lista lor este în documentație). În afară de acestea, pentru a vă defini propriile excepții este necesar să creați o subclasă a clasei `Exception`.

Puteți folosi în loc de blocul *try-except* și keyword-ul with, ca în exemplul din secțiunea Operații cu fișiere. Această construcție apelează automat o metodă de clean-up a obiectului (ex: `close` pentru fișiere) la apariția unei excepții. Această construcție o vom folosi în laboratoarele următoare și pentru obiectele de sincronizare.

Operații cu fișiere

Lucrul cu fișiere este, de asemenea, simplu, iar obiectele și metodele utilizate sunt oferite în modulul io.

Pentru a obține un obiect de tip fișier, se apelează funcția `open`, de obicei cu doi parametri:

- numele (calea) fișierului
- modul de acces

- *r* - read only
- *w* - write only și dacă există va fi suprascris
- *a* - append
- *r+* - citire şi scriere
- *rb*, *wb*, *r+b* - deschide fișierul în mod binar

```
import io
f = open('input.txt','w')
f.write("hello")
f.close()
```

Odată obținut obiectul fișier `f`, se vor putea folosi funcțiile pentru lucrul cu fișiere: `read`, `readline`, `readlines`, `write`, `seek` sau `close` ca metode ale clasei `file` (e.g. `f.read()`).

Spre deosebire de Java, Python nu vă obligă să încadrați codul de lucru cu fișierele într-un bloc try/catch pentru a preveni eventualele excepții. Este recomandat însă să îl încadrați în try/except, sau să folosiți operatorul `with` introdus în Python 2.5, ca în exemplul următor.

```
with open('in.txt', 'r') as f:
    for line in f:   #citire linie cu linie din fisier
        print line
```

Pentru manipularea path-urilor, verificarea fișierelor și directoarelor și alte operații cu acestea puteți folosi funcțiile din modulele os și os.path. Exemplul următor folosește funcțiile de verificare a existenței și tipului unui path.

```
import os, os.path
if not os.path.exists(output_folder):
        os.mkdir(output_folder)
elif not os.path.isdir(output_folder):
    print "Given path is not a folder"
```

Main-ul

Construcția __main__ este opțională în scripturile Python, și rolul său este similar funcțiilor main din alte limbaje (C, Java etc).

Codul conținut într-un script Python este executat și fără prezența acestui mecanism, însa dezavantajul omiterii lui este că atunci când se include fișierul cu `import` se va executa și codul din el, ca în exemplul următor:

utilities.py

```
def f1():
   print "hello1"

f1()
```

mymodule.py

```
import utilities
utilities.f1()
```

```
$ python mymodule.py
hello1
hello1
```

Dacă în modulul utilities apelul funcției f1() se afla într-un `__main__`, aceasta nu se apela, deoarece `__name__` este numele modulului inclus și codul de după `if __name__ == "__main__":` nu se va executa.

```
def f1():
    print __name__
    print "hello1"

if __name__ == "__main__":
    f1()
```

```
$ python mymodule.py
utilities
hello1
$ python utilities.py
__main__
hello1
```

Module utile

random

Generarea de numere pseudo-aleatoare se face la fel de ușor ca și în alte limbaje.

- se include modulul `random` (`import random`)
- se stabilește eventual seed-ul folosind `random.seed`
- se apelează metodele oferite în modul, cum ar fi `random.randint(a,b)`, ce va întoarce un întreg cuprins în intervalul închis `[a,b]`.

pickle

În unele cazuri, pentru a scrie obiecte Python în fișiere, a le stoca în baze de date sau a le transmite pe sockets, acestea trebuie transformate într-un string, proces numit `pickling` în Python (iar procesul invers: `unpickling`). Folosind modulul `pickle` cu funcțiile `pickle.dump` și `pickle.load`,

acest lucru poate fi realizat cu usurință, ca în următorul exemplu:

```
# Salvam un dictionar intr-un fisier folosind pickle
import pickle
culoare_favorita = { "caisa": "galbena", "portocala": "orange", "cireasa":
"rosie" }
print culoare_favorita
pickle.dump(culoare_favorita, open("save.p", "w"))

# Incarcam dictionarul inapoi din fisier folosind pickle
culoarea_mea_fav = pickle.load(open("save.p"))
print culoarea_mea_fav
# culoarea_mea_fav e acum { "caisa": "galbena", "portocala": "orange",
"cireasa": "rosie" }
```

Exerciții

Folosind scheletul de cod implementați un *CoffeeMaker*. Acesta primește comenzi de la utilizator (e.g. să facă o cafea) și afișează rezultatele/mesajele. Comenzile sunt definite în modulul `coffee_maker`. Puteți personaliza mesajele afișate, resursele, puteți alege ce structuri de date să folosiți etc.

- 3p - implementarea tutoror comenzilor din schelet
- 2p - folosirea funcțiilor
- 2p - lucrul cu liste sau/și dicționare
- 2p - folosiți cel puțin o dată o funcție a stringurilor (e.g. strip(), lower(), format()) și un alt mod de a afișare în afară de cel cu concatenare (e.g. interpolare cu %, format())
- 1p - rulați *pylint coffee_maker.py*, observați statisticile și aduceți cel puțin o îmbunatățire code-style-ului (dacă e cazul)
- bonus 2p - citiți rețetele pentru cafea din fișierele oferite în directorul recipes din scheletul de cod. Implementați citirea într-un modul separat (e.g. cel oferit în schelet)

Disclaimer: Recomandăm scrierea în engleză a documentației, comentariilor și a numelor de variabile, funcții, clase, module atât la laborator cât și la teme. Încercăm să vă oferim schelete de cod care să respecte acest lucru.

Folosiți documentația modulelor Python necesare în exerciții! În documentație aveți și exemple de utilizare ale unor metode sau construcții de limbaj.

- string - pentru lucrul cu șiruri de caractere
- Sequence Types - pentru liste și dictionare
- io module - pentru lucrul cu fișiere

Recomandăm parcurgerea acasă și a exercițiilor din anii trecuți sau a materialelor recomandate în secțiunea Referinte pentru a vă familiariza mai bine cu pythonul pentru următoarele două laboratoare

și pentru tema 1.

Resurse

- Responsabilul acestui laborator: Adriana Drăghici
- PDF laborator
- Schelet laborator
- Soluție laborator
- Exercițiile (cu soluții) de anii trecuți (sunt mai multe și abordează concepte mai avansate, de exemplu generatori):
  - 2017, 2016: lab1-2017.zip
  - 2015: lab1-skel-2015.zip, lab1-sol-2015.zip
  - 2014: lab1_exercitii_2014.zip

Referințe

- Documentație Oficială
- Google Python Class
- Style Guide for Python Code
- **Fun Stuff**
  - Real Python - learning Python by example
  - Interesting Features
- **Cărți**
  - Learn Python the hard way
  - Learning Python by Mark Lutz, O'Reilly Media, 5th Edition