Fire de execuție în Python

Obiective

Scopul acestui laborator îl reprezintă familiarizarea cu lucrul cu thread-uri și obiecte de sincronizare în Python. Pentru acestea aveți nevoie de cunoașterea elementelor de sintaxă Python prezentate în laboratorul 1 și de lucrul cu clase prezentat în laboratorul acesta.

Recapitulare laborator 1

La laborator și teme folosim Python 2.6 și 2.7, nu Python 3.x.

Particularități de limbaj:

- **Indentarea** este obligatorie pentru a delimita blocurile de cod.
- Este dynamically și strongly typed:
  - *dynamically typed* - tipurile variabilelor nu se precizează în cod
  - *strongly typed* - pentru că nu se pot face conversii de tip implicite (e.g. adunare de string cu integer)
    - pentru conversiile explicite între tipurile numerice, boolean și șiruri de caractere folosiți funcțiile built-in
- Keywords:
  - `None` este echivalentul null în Java
  - `pass` este echivalentul `{}` din c/java
- Tipurile de date cele mai folosite sunt *int, float, string, boolean, list, tuple, dict*.

Un fișier de cod Python este considerat un **modul**. Pentru a folosi alte module utilizăm `import` în următoarele modalități:

import_example.py

```
import random
random.randint(0,4)    # trebuie specificat numele modulului

from random import *   # import tot continutul modulului
randint(0,4)           # nu mai trebuie specificat numele modului

from random import randint  # import doar randint
randint(0,4)

import random as rand  # folosire alias pentru numele modulului
```

```
rand.randint(0,4)
```

Construcția **`if __name__ == "__main__"`** delimitează 'main'-ul unui modul. Acesta nu este obligatoriu însă dacă nu e folosit, codul pe care l-am fi delimitat în el s-ar executa de fiecare dată când se face import la modulul respectiv.

Funcțiile se declară folosind keyword-ul `def` și nu li se specifică tip de return sau tipuri pentru parametri. Se poate simula supraîncărcarea (overloading) metodelor folosind parametri default.

func_example.py

```
def f(a, b="", c=0):
    print " ".join([a, b, str(c)])

f("hello")                  # hello  0
f("hello", "lab")           # hello lab 0
f("hello", c=2)             # hello  2
f("hello", "lab", c=2)      # hello lab 2
```

Ce este un thread?

Sistemele de calcul moderne sunt capabile de a executa mai multe operatii în acelasi timp. Spre exemplu, un utilizator casnic poate să utilizeze un procesor de text în timp ce, pe același calculator, alte aplicații descarcă fișiere, controlează o coada de taskuri pentru imprimantă sau rulează un fișier video. În exemplul anterior sistemul de operare este cel care permite rularea mai multor aplicații simultan, dar această idee se poate extinde și la nivelul unei aplicații. O aplicație ce rulează un stream video online trebuie simultan să *citească* conținutul video de pe rețea, să îl *decompreseze*, să *actualizeze* display-ul local cu aceste informații etc. Spunem că aplicațiile ce oferă aceste capabilități constituie un software concurent.

Deci ce este concurența? **Concurența** este proprietatea unei logici de program de a putea executa **simultan** un set de task-uri. **Paralelismul** reprezintă o metodă de implementare a acestei paradigme de programare ce permite rularea unui set de task-uri într-un mod care utilizează core-uri multiple, procesoare multiple sau chiar mai multe mașini (într-o structură de tip cluster de exemplu).

Revenind la thread-uri, acestea reprezintă o metodă specifică de implementare a concurenței. *Thread-urile* reprezintă fire de execuție create (*spawned*) în cadrul unui program principal (*process*) ce execută concurent task-uri definite de programator. Implementarea threadurilor diferă de la un sistem de operare la altul, dar, în cele mai multe cazuri, un fir de execuție este parte a unui proces. Mai multe thread-uri pot exista în cadrul aceluiași proces, ele partajând anumite resurse: memorie, descriptori I/O etc. În această privință thread-urile diferă de procese prin faptul că variabilele globale pot fi accesate de către toate thread-uri unui proces și pot servi ca mediu de comunicație între thread-uri. Fiecare thread are totuși și un set propriu de variabile locale. Din acest motiv thread-urile mai sunt numite și *lightweight processes*.

În cadrul unui sistem uni-procesor, rularea concurentă a mai multor fire de execuție se face prin

metoda *partajării timpului de execuție* (*time sharing / time division / time slicing*), sistemul de operare alternând succesiv între execuția thread-urile active (percepția este cea a rulării simultane însă în realitate un singur thread rulează la un moment dat).

În cadrul unui sistem multi-procesor sau multi-core, thread-urile vor rula în general cu adevărat simultan, cu fiecare procesor rulând un thread specific.

Din punct de vedere al suportului pentru programarea multithreading limbajele se împart în două categorii:

- limbaje cu thread-uri utilizator (*green threads*) ce nu sunt vizibile sistemului de operare, ci doar la nivelul unui singur proces (găsiți vreun dezavantaj?)
- limbaje cu thread-uri native (adesea denumite și *kernel threads*) ce sunt vizibile la nivelul sistemului de operare și care permite execuția lor paralelă

Clase și obiecte în Python

Trebuie subliniat că în Python, cuvântul "obiect" nu se referă neapărat la instanța unei clase. Clasele în sine sunt obiecte, iar, în sens mai larg, în Python toate tipurile de date sunt obiecte. Există tipuri de date care nu sunt clase: numerele întregi, listele, fișierele.

Un obiect se crează în Python prin folosirea cuvantului cheie `class`, ca în exemplul de mai jos, în care se crează un obiect de tip *class* cu numele *className*. *Interiorul clasei* poate conține definiții de metode sau clase și atribuiri pentru variabile. Clasa este derivată din *super_class1* și din *super_class2*. Spre deosebire de Java, numele modulului nu trebuie să fie la fel cu al vreunei clase definite în el.

```
class className [(super_class1 [, super_class2]*)]:
    [interiorul clasei]
```

Clasele suportă **multiple inheritence** și nu există un contract propriu-zis pentru interfețe. Pentru a crea clase abstracte există modulul abc (Abstract Base Classes). Pentru metodele pe care vreți să le considerați abstracte puteți transmite excepția `Not Implemented Error` sau să adăugați în corpul funcției doar keyword-ul `pass`.

În lucrul cu clase, trebuie avute în vedere următoarele reguli:

- Primul argument pentru metodele unei clase este întotdeauna obiectul sursă, numit **`self`**.
- Când ne referim la membrii clasei, trebuie să folosim `self.membru`, într-un mod asemanator cu folosirea "this" din Java (doar că în Python este obligatoriu de folosit `self` nu doar pentru a face distincție între variabilele clasei și parametrii sau variabilele cu același nume din funcții)
- Metoda specială `__init__()` este apelată la instanțierea clasei și poate fi considerată un **constructor**. Definirea metodelor `__init__()` este opțională.
- Metoda specială `__del__()` este apelată când nu mai sunt referințe la acest obiect și poate fi asemuită cu un **destructor**. Definirea metodelor `__del__()` este opțională.
- În cazul moștenirii, în metoda `__init__()` trebuie întâi apelat `__init__()`-ul claselor părinte.
- Implicit toate variabilele si metodele claselor sunt publice, pentru mai multe detalii citiți aici.
- Instanțierea se face prin apelarea obiectului clasă, posibil cu argumente. O altă modalitate este folosirea funcției built-in apply.

class_example.py

```
class Student:
    """  O clasa care reprezinta un student. Comentariile docstring se
pun asa :) """
    def __init__(self, name, grade=5):   # parametrii default
(echivalent cu a avea mai multi constructori overloaded)
        self.name = name                  # variabilele clasei pot fi
declarate oriunde!
        self.change_grade(grade)

    def change_grade(self, grade):        # primul parametru este 'self'
        self.grade = grade

x = Student("Alice")
y = Student("Bob", 10)
x.change_grade(8)
```

Clasele Python pot avea membri statici. În cazul variabilelor este suficient să nu se folosească “self”, iar pentru metode avem două variante, una folosind decoratorul @staticmethod, alta cealaltă folosind funcția built-in staticmethod.

static_example.py

```
class Util:

    x = 2      # variabila statica

    @staticmethod
    def do_stuff():
        print "stuff"

    def do_otherstuff():
        print "other stuff"

    # a alta metoda de a face o functie statica:
    do_otherstuff = staticmethod(do_otherstuff)

Util.do_stuff()
Util.do_otherstuff()
print Util.x
```

Clase Python pe scurt:

- trebuie să folosiți `self.nume_funcție` sau `self.nume_variabila`
- o clasă poate moșteni mai multe clase
- `__init__()` este numele constructorului. Puteți avea un singur constructor, pentru a simula mai

mulți contructori folosiți parametri default.
- puteți avea metode și variabile statice
- nu aveți access modifiers
- instanțiere: `nume_instanta = NumeClasa(argumente_constructor)`

Programare concurentă în Python

În Python, programarea concurentă este suportată prin intermediul modulului threading. Acest modul oferă clasa Thread, care permite crearea și managementul thread-urilor, precum și o serie de clase (Condition, Event, Lock, RLock, Semaphore) care oferă facilități de sincronizare și comunicare între thread-urile unui program Python.

Thread-uri

Un fir de execuție concurentă este reprezentat în Pyhton de clasa *Thread*. Cel mai simplu mod de a specifica instrucțiunile care se doresc a fi rulate concurent este de a apela constructorul lui *Thread* cu numele unei funcții care conține aceste instrucțiuni precum în exemplul următor. Pornirea thread-ului se face apoi cu metoda *start()*, iar pentru a astepta terminarea execuției thread-ului se folosește metoda *join()*.

exemplul1.py

```
from threading import Thread

""" Functie care vrem sa ruleze concurent """
def my_concurrent_code(nr, msg):
    print "Thread", nr, "says:", msg

# creeaza obiectele corespunzatoare thread-urilor
t1 = Thread(target = my_concurrent_code, args = (1, "hello from thread"))
t2 = Thread(target = my_concurrent_code, args = (2, "hello from other
thread"))

# porneste thread-urile
t1.start()
t2.start()

# executia thread-ului principal continua de asemenea
print "Main thread says: hello from main"

# asteapta terminarea thread-urilor
t1.join()
t2.join()
```

Se folosește parametrul *target* al constructorului pentru a pasa numele functiei concurente și, opțional, pot fi folosiți parametrii *args* sau *kwargs* pentru a specifica argumentele funcției concurente, dacă ele există. *args* este folosit pentru a trimite argumentele funcției concurente ca un tuplu, iar *kwargs* este folosit pentru a trimite argumentele ca un dictionar.

Pentru a diferenția un tuplu cu un singur element de folosirea obișnuită a parantezelor se utilizează următoarea sintaxă:

```
# t contine int-ul 42
t = (42)
# t contine un tuplu cu un singur element
t = (42,)
```

Crearea unui obiect Thread nu pornește execuția firului de execuție. Acestu lucru se întâmplă doar după apelul metodei *start()*.

O metodă alternativă de a specifica instrucțiunile care se doresc a fi rulate concurent este de a crea o subclasă a lui *Thread* care suprascrie metoda *run()*. Se poate de asemenea suprascrie și metoda *__init__()* (constructorul) pentru a primi argumentele cu care vor fi initializate câmpurile proprii subclasei. Dacă optați pentru această abordare nu este indicat să suprascrieți alte metode ale clasei *Thread*, decât constructorul și *run()*.

exemplul2.py

```
from threading import Thread

""" Clasa care incapsuleaza codul nostru concurent """
class MyThread(Thread):
    def __init__(self, nr, msg):
        Thread.__init__(self)
        self.nr = nr
        self.msg = msg

    def run(self):
        print "Thread", self.nr, "says:", self.msg

# creeaza obiectele corespunzatoare thread-urilor
t1 = MyThread(1, "hello from thread")
t2 = MyThread(2, "hello from other thread")

# porneste thread-urile
t1.start()
t2.start()

# executia thread-ului principal continua de asemenea
print "Main thread says: hello from main"
```

```
# asteapta terminarea thread-urilor
t1.join()
t2.join()
```

La suprascrierea constructorului clasei *Thread* nu uitați să apelați și constructorul clasei de bază.

Pe lângă clasa *Thread* și clasele de sincronizare, modulul *threading* mai conține și o serie de funcții utile în debugging-ul programelor cu mai multe fire de execuție:

- Funcția *active_count()* returnează numărul curent de thread-uri active (care rulează).
- Funcția *current_thread()* returnează obiectul *Thread* corespunzător firului de execuție care a rulat apelul funcției. Acest obiect poate fi folosit pentru a afișa informații despre thread-ul curent, cum ar fi numele acestuia. Numele unui thread este implicit “Thread-N” (unde N este un număr unic), dar poate fi ușor schimbat prin folosirea parametrului *name* al constructorului clasei *Thread*.
- Funcția *enumerate()* returnează o listă cu toate obiectele *Thread* active.

Interpretorul cel mai popular de Python (CPython) folosește un lock intern (GIL - Global Interpreter Lock) pentru a simplifica implementarea unor operații de nivel scăzut (managementul memoriei, apelul extensiilor scrise în C etc.) Acest lock permite execuția unui singur thread în interpretor la un moment dat și limitează paralelismul și performanța thread-urilor Python. Mai multe detalii despre GIL puteți găsi în această prezentare.

Elemente de sincronizare

Pentru ca un program concurent să funcționeze corect este nevoie ca firele sale de execuție să coopereze în momentul în care vor să acceseze date partajate. Această cooperare se face prin intermediul partajării unor elemente de sincronizare care pun la dispoziție un API ce oferă anumite garanții despre starea de execuție a thread-urilor care le folosesc.

Thread

Pe lângă facilitățile de creare a noi fire de execuție, obiectele de tip *Thread* reprezintă și cele mai simple elemente de sincronizare, prin intermediul metodelor *start()* și *join()*.

Metoda *start()* garantează că toate rezultatele thread-ului care o apelează (să-l numim *t1*), până în punctul apelului, sunt disponibile și în thread-ul care va porni (sa-l numim *t2*). A se observa că nu se oferă nici un fel de garanție despre rezultatele lui *t1* care urmează după apel. *t2* nu poate face nici o presupunere în acest caz, fără a folosi alte elemente de sincronizare.

Metoda *join()* garantează thread-ului care o apelează (să-l numim *t1*) că thread-ul asupra căreia este apelată (să-l numim *t2*) s-a terminat și nu mai accesează date partajate. În plus toate rezultatele lui *t2* sunt disponibile și pot fi folosite de către *t1*. A se observa că, față de metoda *start()*, metoda *join()*

blochează execuția thread-ului care o apelează (*t1*) până când *t2* își termină execuția. Spunem că *join()* este o metodă blocantă.

Lock

Lock-ul este un element de sincronizare care oferă acces exclusiv la porțiunile de cod protejate de către lock (cu alte cuvinte definește o secțiune critică). Pyhton pune la dispoziție clasa *Lock* pentru a lucra cu acest element de sincronizare. Un obiect de tip *Lock* se poate afla într-una din următoarele două stări: **blocat** sau **neblocat**, implicit, un obiect de tip *Lock* fiind creat în starea **neblocat**. Sunt oferite două operații care controlează starea unui lock: *acquire()* și *release()*.

Metoda *acquire()* va trece lock-ul în starea blocat. Dacă lock-ul se afla deja în starea blocat, thread-ul care a apelat *acquire()* se va bloca până când lock-ul este eliberat (pentru a putea fi blocat din nou). Metoda *release()* este cea care trece lock-ul în starea deblocat. Cele două metode garantează că un singur thread poate deține lock-ul la un moment dat, oferind astfel posibilitatea ca un singur thread să execute secțiunea de cod critică. O altă garanție a lock-ului este că toate rezultatele thread-ului care a efectuat *release()* sunt disponibile și pot fi folosite de următoarele thread-uri care execută *acquire()*.

Spre deosebire de un mutex (ex: pthread_mutex), în Python, metodele *acquire()* și *release()* pot fi apelate de thread-uri diferite. Cu alte cuvinte un thread poate face *acquire()* și alt thread poate face *release()*. Datorită acestei diferențe subtile nu este recomandat să folosiți un obiect *Lock* în acest mod. Pentru a reduce confuziile și a obține același efect se poate folosi un obiect *BoundedSemaphore* inițializat cu valoarea 1.

Lock-ul este utilizat în majoritatea cazurilor pentru a proteja accesul la structuri de date partajate, care altfel ar putea fi modificate de un fir de execuție în timp ce alte fire de execuție încearcă simultan să citească sau să modifice și ele aceeași structură de date. Pentru a rezolva această situație, porțiunile de cod care accesează structura de date partajată sunt încadrate între apeluri *acquire()* și *release()* pe **același** obiect *Lock* partajat de toate thread-urile care vor sa acceseze structura.

Exemplul de mai jos prezintă folosirea unui lock pentru a proteja accesul la o listă partajată de mai multe thread-uri.

exemplul3.py

```
from threading import Lock, Thread

def inc(lista, lock, index, n):
    """ Incrementeaza elementul index din lista de n ori """
    for i in xrange(n):
        lock.acquire()
        lista[index] += 1
        lock.release()


def dec(lista, lock, index, n):
```

```
    """ Decrementeaza elementul index din lista de n ori """
    for i in xrange(n):
        lock.acquire()
        lista[index] -= 1
        lock.release()

# lista si lock-ul care o protejeaza
my_list = [0]
my_lock = Lock()

# thread-urile care modifica elemente din lista
t1 = Thread(target = inc, args = (my_list, my_lock, 0, 100000))
t2 = Thread(target = dec, args = (my_list, my_lock, 0, 100000))

# lista inainte de modificari
print my_list

t1.start()
t2.start()

t1.join()
t2.join()

# lista dupa modificari
print my_list
```

Puteți folosi construcția *with* pentru a delimita o secțiune critică astfel:

```
def inc(lista, lock, index, n):
    for i in xrange(n):
        with lock:
            lista[index] += 1
```

Semaphore

Semaforul este un element de sincronizare cu o interfață asemănătoare Lock-ului (metodele *acquire()* și *release()*) însă cu o comportare diferită. Python oferă suport pentru semafoare prin intermediul clasei *Semaphore*.

Un *Semaphore* menține un contor intern care este decrementat de un apel *acquire()* și incrementat de un apel *release()*. Metoda *acquire()* nu va permite decrementarea contorului sub valoarea 0, ea blocând execuția thread-ului în acest caz până când contorul este incrementat de un *release()*. Metodele *acquire()* și *release()* pot fi apelate fără probleme de thread-uri diferite, această utilizare fiind des întâlnită în cazul semafoarelor.

Un exemplu clasic de folosire a semaforului este acela de a limita numărul de thread-uri care accesează concurent o resursă precum în exemplul următor:

exemplul4.py

```
from random import randint, seed
from threading import Semaphore
from time import sleep

def access(nr, sem):
    sem.acquire()
    print "Thread-ul", nr, " acceseaza"
    sleep(randint(1, 4))
    print "Thread-ul", nr, " a terminat"
    sem.release()

# initializam semaforul cu 3 pentru a avea maxim 3 thread-uri active la
un moment dat
semafor = Semaphore(value = 3)

# stocam obiectele Thread pentru a putea face join
thread_list = []

seed()

# pornim thread-urile
for i in xrange(10):
    thread = Thread(target = access, args = (i, semafor))
    thread.start()
    thread_list.append(thread)

# asteptam terminarea thread-urilor
for i in xrange(len(thread_list)):
    thread_list[i].join()
```

Exerciții

1. *Hello Thread* - rezolvați exercițiile din fișierul `task1.py` din scheletul de laborator. (**4p**)
2. Protejarea variabilelor folosind locks - rezolvați exercițiile din fișierul `task2.py` din scheletul de laborator. (**3p**)
3. Implementați problema producator-consumator folosind semafoare. (**3p**)
   - Mai mulți producatori și mai multi consumatori comunică printr-un buffer partajat, limitat la un numar fix de valori. Un producător pune câte o valoare in buffer iar un consumator poate să ia câte o valoare din buffer.
   - Aveți nevoie de două semafoare, unul pentru a indica dacă se mai pot pune valori în buffer și celălalt pentru a arata dacă există vreo valoare care poate fi luată din buffer de către consumatori.

4. Implementați problema filosofilor. (**2p**)
   - Se consideră mai mulți filozofi ce stau în jurul unei mese rotunde. În mijlocul mesei este o farfurie cu spaghete. Pentru a putea mânca, un filozof are nevoie de două furculițe. Pe masă există câte o furculița între fiecare doi filozofi vecini. Regula este ca fiecare filozof poate folosi furculițele din imediata sa apropriere. Trebuie evitată situația în care nici un filozof nu poate acapara ambele furculițe).

Resurse

- Responsabilii acestui laborator: Adriana Drăghici, Dan Dragomir
- PDF laborator
- Schelet laborator

Referințe

Documentație module

- modulul thread
- modulul threading - Thread, Lock, Semaphore

Detalii legate de implementare

- Implementarea obiectelor de sincronizare în CPython
- Python Threads and the Global Interpreter Lock
- Understanding the Python GIL (prezentare foarte bună și amuzantă)

Despre concurență și obiecte de sincronizare

- Introduction to semaphores (video)
- Understanding Threading in Python
- Little book of semaphores
- Programming on Parallel Machines (Chapter 3)