Corso di Linguaggi Dinamici
Esercitazione 2 – moduli e classi

Esercizio 1. Viene richiesta la scrittura di un programma ad oggetti in Python che rappresenti un piccolo insieme di animali. Il programma deve essere modularizzato in file distinti, scrivendo un file/modulo per ciascuna delle seguenti classi.

Tutti gli attributi delle classi (es. nome ed età per Animale) devono essere implementati con attributi privati che saranno acceduti soltanto tramite i relativi metodi get/set.

Classe Animale (da cui erediteranno tutte le altre)

- Classe **Animale**:
  - Attributi:
    - stringa: nome
    - intero: età
  - Metodi:
    - get/set per nome ed età
    - dorme (che accetta un intero n e si ferma per n secondi;
      se n non è specificato, si ferma di default per 1 secondo;
      ritorna il numero n di secondi per cui si è fermato)

- Classe **Cane**: sottoclasse di Animale
  - Attributi:
    - stringa: razza
  - Metodi:
    - get/set per razza
    - info (che ritorna la razza)
    - parla (che ritorna la stringa “abbaia”)
    - si_muove (che ritorna la stringa “corre”)
    - mangia (che ritorna la stringa “mangia”)
    - beve (che ritorna la stringa “beve”)

- Classe **Cavallo**: sottoclasse di Animale
  - Attributi:
    - stringa: mantello
  - Metodi:
    - get/set per mantello
    - info (che ritorna il mantello)
    - parla (che ritorna la stringa “nitrisce”)
    - si_muove (che ritorna la stringa “galoppa”)
    - mangia (che ritorna la stringa “mangia”)
    - beve (che ritorna la stringa “beve”)

- Classe **Leone**: sottoclasse di Animale
  - Attributi:
    - intero: peso
  - Metodi:
    - get/set per peso
    - info (che ritorna il peso)
    - parla (che ritorna la stringa “ruggisce”)
    - si_muove (che ritorna la stringa “va a caccia”)
    - mangia (che ritorna la stringa “divora”)
    - beve (che ritorna la stringa “beve”)

Si scriva poi un programma principale **zoo.py** che dovrà generare un numero intero casuale tra 1 a 10 di oggetti per ciascuna specie (Cane, Cavallo, Leone), assegnando nomi ed età scelti a caso. Successivamente, dovrà implementare un ciclo di 20 iterazioni: per ogni iterazione, dovrà scegliere a caso uno degli oggetti generati precedentemente ed una azione (sempre scelta a caso) tra parla(), si_muove(), mangia(), beve() e dorme(n).
Per ogni iterazione, si deve stampare una stringa che riporti il nome dell'animale, la sua età, le informazioni specifiche (attraverso l'invocazione del metodo info()), e il risultato dell'invocazione dell'operazione scelta. Nel caso dell'operazione dorme(n), dopo l'invocazione deve stampare la stringa "ha dormito per n secondi", dove n è il numero di secondi che deve essere compreso tra 1 e 3. Dopo aver stampato la stringa, il programma si deve fermare per 1s ed il ciclo ricomincia.

Suggerimenti:

Per effettuare la scelta casuale dei nomi degli animali, posso creare liste di nomi e scegliere un indice a caso tra 0 e len(lista)-1.

Per la generazione di numeri casuali si faccia riferimento alla funzione **random()**
>>> pydoc random
>>> from random import random
random() restituisce valori float nell'intervallo semi-aperto [0.0, 1.0)

Per mettere in attesa il programma, si faccia riferimento alla funzione **sleep()** del modulo **time**
>>> pydoc time
>>> from time import sleep
>>> sleep(n) #n intero

Corso di Linguaggi Dinamici
Esercitazione 3 – classi iterabili

Esercizio 1. Viene richiesta la scrittura di un programma in Python che definisce una **classe** che rappresenta un **oggetto iterabile**. In particolare, si richiede che la classe accetti come parametro di ingresso per il costruttore un **dictionary**, che rappresenta una mini anagrafica: i valori di **chiave** sono **stringhe** che rappresentano il nominativo delle persone incluse, e i valori associati sono (ad esempio) i loro numeri di telefono.
Si richiede che la classe sia in grado di **iterare** sugli elementi del dictionary (offrendo i metodi __iter__() e next()) in modo da restituire un elemento alla volta del dictionary **riordinato secondo l'ordine alfabetico delle chiavi.**

A livello funzionale, si richiede quindi il seguente comportamento:

```
>>> for item in OrdDict({'Canali':6317,'Rossi':4527, 'Botti':3624, 'Baraldi':6352, 'Amedei':3652}):
...         print item
{'Amedei': 3652}
{'Baraldi': 6352}
{'Botti': 3624}
{'Canali': 6317}
{'Rossi': 4527}
```

Trovate la traccia impostata nel file **Ordered_dictionary.py**

Corso di Linguaggi Dinamici
A.A. 2012/13
Esercitazione sui file di Python

Esercizio 1. Scrivere un programma che definisca una funzione ricorsiva **splitFully.** La funzione prende in ingresso un **path** e, richiamandosi ricorsivamente ed utilizzando la funzione **os.path.split()**, separa tutte le componenti del path del file system passatole come parametro e le ritorna in una lista.
Ad esempio: l'invocazione splitFully('/usr/local/bin')
ritorna come risultato
['/', 'usr', 'local', 'bin']

Suggerimenti:

- La funzione split(path) ritorna una stringa nulla come secondo elemento della tupla quando path è composto di un solo elemento
  >>> os.path.split("/")
  ('/', '')
- E' possibile comporre liste con l'operatore +
  >>> a = [1,3,5] + [7,9]
  >>> a
  [1,3,5,7,9]

Esercizio 2. Scrivere un programma che definisca una funzione **printDir(path)** che visualizzi il contenuto completo di una cartella
del file system passata come parametro di ingresso **path**. In pratica, si richiede di scrivere il **percorso assoluto** di tutti i file e le cartelle contenute in **path**.
Poi, scrivere una variante **printTree(path)** che visualizzi ricorsivamente anche il contenuto delle eventuali sotto-cartelle presenti.
Suggerimenti: si implementino 2 versioni per il programma con **printTree(path)** : una che utilizza la funzione walk() e l'altra che non la usa, ma si limita ad usare le funzioni listdir() e isdir()

Esercizio 3. Scrivere un semplice programma per la gestione di un blog.
In particolare, supponiamo che ogni post del blog sia memorizzato sotto forma di una tupla binaria formata da due stringhe, nel seguente modo: ('data_ora', 'testo'). Il blog nel suo complesso è quindi rappresentato come una **lista di tuple**, dove ogni tupla è un post, ed è memorizzato sul file "diary-data.pickle" attraverso il protocollo **pickle**.
Si richiede di scrivere 2 programmi in Python:

1) blogAdd.py, che dovrà leggere il contenuto del file "diary-data.pickle" ed effettuarne l'unpickling in una lista. Se il file non esiste, dovrà creare una lista vuota per i prossimi post. Quindi, dovrà leggere la data e l'ora di sistema e metterla in una stringa 'data', poi richiedere un nuovo post all'utente e mettere il messaggio inserito in una stringa 'testo'; dovrà poi aggiungere la tupla ('data', 'testo') alla lista ed effettuare il pickling della lista nel file "diary-data.pickle"
2) blogPrint.py, che leggerà invece il contenuto del file per effettuarne la stampa.

Suggerimenti: utilizzare i moduli **pickle** e **time** (ad es.: data = time.ctime(time.time()) - ctime() ritorna direttamente una stringa).

Corso di Linguaggi Dinamici
Esercitazione sulla documentazione in Python

Esercizio 1. Si richiede la creazione di un **package zoo** contenente il modulo **Animale.py** (e il file **__init__.py** per l'inizializzazione). Quindi, si richiede la creazione automatica della documentazione in formato **sphinx HTML** per il package **zoo**, comprensiva del modulo **Animale** e di metodi/classi contenuti. Si utilizzino le funzionalità **autodoc di sphinx**.
Si richiede che la documentazione generata sia identica a quella riportata nell'esempio **index.html**

Suggerimenti
Per la generazione del file **Animale.py** adeguatamente commentato, si parta dal file omonimo nella cartella **basic/** del codice sulla documentazione in Python.

Corso di Linguaggi Dinamici
Esercitazione sullo unit testing in Python

Esercizio 1. Si richiede la scrittura di una **suite** di test per il programma ad oggetti **zoo**. Tale suite dovrà contenere due suite di test distinte create per testare le classi **Animale** e **Cane**. In particolare, per ogni classe considerata, dovranno essere esaminati i seguenti aspetti:

- consistenza architetturale in fase di creazione di un oggetto istanza della classe
- esistenza dei metodi invocabili get()/set()
- congruità dei risultati dei metodi get()/set() su input validi (stringhe/interi non nulli)
- esistenza degli altri metodi invocabili previsti
- congruità dei risultati degli altri metodi su input validi (stringhe/interi non nulli)
- robustezza con dati invalidi (sollevamento di eccezioni inserite)

Suggerimenti:
Si utilizzi una fase di setup per creare un oggetto istanza della classe da testare.
Usare le funzioni *isinstance()* e *callable()* per verificare rispettivamente se un oggetto è istanza di una classe e se un metodo esiste.

Corso di Linguaggi Dinamici
Esercitazione sulle opzioni

Esercizio 1. Si richiede di rendere il programma ad oggetti **zoo.py (zoo_opt.py) configurabile da linea di comando.** In particolare, il programma deve consentire di specificare attraverso opzioni il numero di oggetti appartenenti ad ogni sottoclasse che devono essere generati.
Es. Opzioni **-c --cane** per la sottoclasse Cane, **-v --cavallo** per Cavallo e **-l --leone** per Leone (seguite da un numero intero che specifica il numero di oggetti).
Per le sottoclassi per cui l'opzione non viene specificata da riga di comando si dovrà generare un numero di oggetti random tra 1 e 10 (default).
Si preveda inoltre la possibilità di passare al programma una opzione **-v --verbose** che, se presente, attivi alla fine del programma una stampa dei nomi e delle età di tutti gli animali generati e inseriti nella lista **zoo[]**.

Corso di Linguaggi Dinamici
A.A. 2012/2013
Esercitazione su configurazione e logging

Esercizio 1. Si introduca una funzionalità di configurazione da file di testo nell'applicazione *zoo.py*. A tal scopo, si faccia uso del modulo software **ConfigParser**. Il file ***zoo.ini*** rappresenta la configurazione tipo da implementare. Si noti che, per semplificare l'implementazione, gli attributi di tutte le sottoclassi sono stringhe e allo stesso modo tutte le azioni invocate sulle sottoclassi ritornano stringhe (inclusa l'azione **dorme(sec)**).
Partire dalla nuova traccia **zoo.py** che richiama il modulo **ClassZoo.py**, il quale implementa lo zoo come una classe che mantiene al suo interno la struttura dati **zoo[]** con gli oggetti generati per ogni sottoclasse.
La classe **ClassZoo** mette a disposizione un metodo **simula()** che seleziona un oggetto a caso dalla lista **zoo[]** e una azione a caso tra le possibili, ritornando una tupla (bestia, id_azione) con l'oggetto selezionato e l'indice identificativo dell'azione.
La classe mette inoltre a disposizione un metodo **build_output(bestia, id_azione)** che prende in ingresso un oggetto e un identificativo di azione: il metodo costruisce la frase che contiene nome, età, attributo e stringa tornata dall'azione invocata e la stampa a video.

Esercizio 2. Si integri nella soluzione precedente un **logger** per l'applicazione ***zoo.py***. Il file ***zoo.log*** rappresenta il log tipo da implementare, di cui segue un estratto:
2013-05-02 19:15:17,949 logger:Cane level:INFO Generato animale di nome Poldo anni 2 razza
2013-05-02 19:15:17,949 logger:Cavallo level:INFO Generato animale di nome Strike anni 2 razza
2013-05-02 19:15:17,954 logger:Zoo level:DEBUG filename:/mnt/dati/Linguaggi-dinamici/linguaggi_dinamici-1213/Esercitazioni/esercitazione_PyA4/ZooAlgo.py function:simula line:103 Selezionato id_zoo=1, id_azione=3, bestia=Stella
2013-05-02 19:15:17,954 logger:Zoo level:INFO Stella (eta' 14 anni, razza Lituano) si abbevera
2013-05-03 09:19:49,542 logger:Cavallo level:DEBUG filename:/mnt/dati/Linguaggi-dinamici/linguaggi_dinamici-1213/Esercitazioni/esercitazione_PyA4/Animale.py function:dorme line:33 Generato tempo di riposo t=5
2013-05-03 09:19:49,542 logger:Zoo level:INFO Stella (eta' 18 anni, razza Arabo) ha dormito per 5 unita' di tempo
Come appare dal file zoo.log, si richiede che il programma faccia uso di diversi logger e di diversi tipi di messaggi di log.

Corso di Linguaggi Dinamici
A.A. 2012/2013
Esercitazione su Pyro

Esercizio 1. Si estenda l'applicazione **zoo.py** con un meccanismo per la distribuzione remota delle classi degli animali. In particolare, si richiede che le classi **Animale, Cane, Cavallo** e **Leone** siano collocate su un server, dove saranno accedute per la fase di generazione degli oggetti. Si noti che gli oggetti relativi ad ogni sottoclasse devono essere generati e mantenuti sul server, mentre al client deve essere passato soltanto il **proxy** di ogni oggetto, che sarà usato per invocarne le funzioni richieste.

Suggerimenti:
In questo caso la lista **zoo[]** sarà una lista contenente i **proxy** degli oggetti generati.

Per lo sviluppo della soluzione, si tenga in considerazione l'esempio **factory**, che è preso (e lievemente adattato) dalla directory fornita dal pacchetto `pyro-examples` /usr/share/doc/pyro-examples/docs/examples, contenente esempi pronti e funzionanti.

Esempio factory (dal file Readme.txt):
This example shows the 'factory' feature of Pyro: it shows how to create new objects on the server and pass them back to the client (actually, not the objects themselves but proxies are passed back).

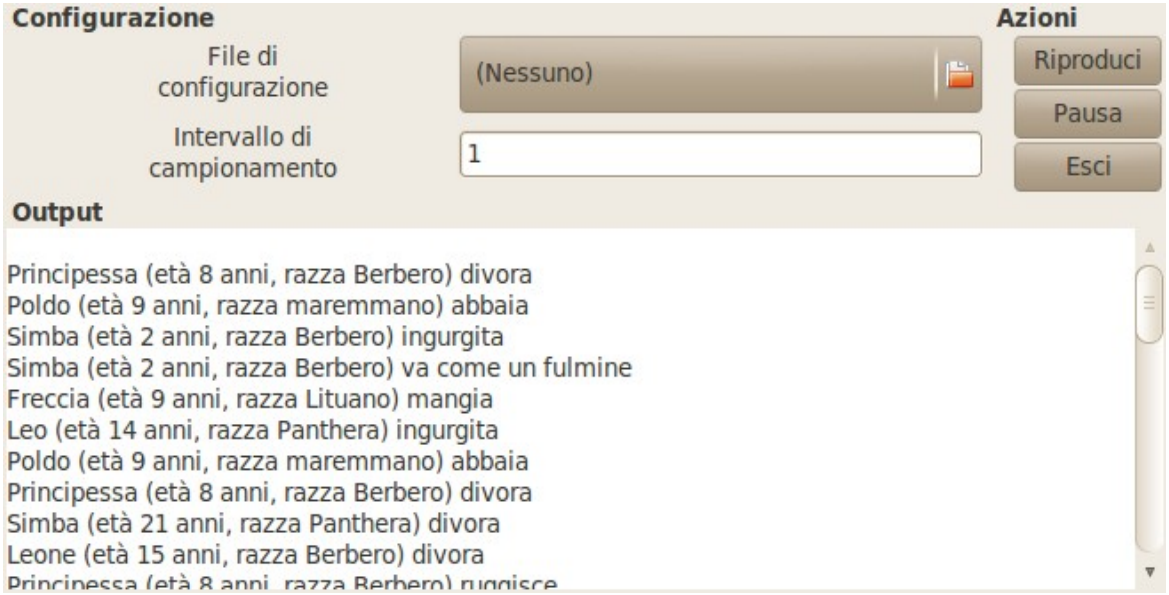
Corso di Linguaggi Dinamici
A.A. 2012/2013
Esercitazione su thread (Queue e thread temporizzati)

Esercizio 1. Scrivere un programma che faccia uso di thread temporizzati per implementare la “rotazione” delle versioni di un file di log di nome “web.log”. Si supponga che il MainThread scriva continuamente sul file di log, e utilizzi un thread temporizzato per effettuare periodicamente (es. ogni 3 sec) un backup del file, per evitare che il log assuma dimensioni troppo grandi.
Ad ogni backup periodico, il thread controllerà l’esistenza del file e, se esistente, rinominerà il file “web.log.1”. Ad una seconda invocazione, il programma dovrà far spazio al backup di un nuovo “web.log” facendo ruotare le precedenti versioni. In altre parole, dovrà rinominare “web.log.1” in “web.log.2” e poi il nuovo “web.log” in “web.log.1”. E così via...

Esercizio 2. Scrivere un programma che faccia uso di una coda Queue per far comunicare più produttori e un consumatore. Si supponga che i produttori e il consumatore siano tutti thread paralleli lanciati da un MainThread. Ciascun produttore (realizzato con una estensione della classe Thread) riceve in ingresso la Queue condivisa e il numero di elementi da generare: ciascun elemento sarà un stringa formata dal nome del thread produttore seguita da un indice numerico che identifica l'elemento. Il consumatore estrae dalla coda ciascun elemento e lo stampa a video.

Corso di Linguaggi Dinamici
A.A. 2012/2013

Esercizio Si sviluppi una **GUI multi threaded** per l'applicazione *zoo.py*. Il layout della applicazione è mostrato in figura.



Il tasto “Riproduci” avvia una riproduzione continua delle frasi di output nella TextView con cadenza determinata dal valore dell'intervallo di campionamento. Il tasto “Pausa” interrompe la riproduzione. Il tasto “Esci” (o, equivalentemente, il bottone di distruzione della finestra principale) terminano l'applicazione. Il dialogo di selezione file permette di impostare un nuovo file di configurazione e di reinizializzare lo stato interno dello zoo. Infine, l'intervallo di campionamento può essere modificato dinamicamente (anche durante la riproduzione) attraverso la casella di testo opportuna.

Suggerimenti:

- Si parta dalla versione di zoo.py dotata di GUI.
- Si utilizzi una coda Queue per la comunicazione tra i processi
- Si consideri che ZooAlgo va considerato il thread produttore: gli elementi prodotti sono le frasi generate e restituite da ZooAlgo.build_output, che verranno messe nella coda. Quindi, ZooAlgo va implementato come un thread.
- ZooGUI è invece il thread che si occupa dell'interfaccia grafica e delle operazioni di consumatore, cioè leggere dalla coda una frase alla volta e aggiornare la textView in output
- Si noti che l'operazione del consumatore è minimale (sola lettura) e deve avvenire periodicamente con periodo uguale all'intervallo di campionamento
- Si osservi la struttura del file zoo.py

NOTE

NOTE

NOTE