Esercizi dicembre 2020

Programmazione Avanzata

a.a. 2020-21

1

Esercizio 1

- Immaginiamo che un pacco venga inviato all'ufficio postale: il pacco puo` essere ordinato, poi spedito all'ufficio postale e quindi ricevuto dal destinatario.
- Vogliamo scrivere il suo stato ogni volta che questo cambia: lo stato iniziale e` ordinato. La classe Pacco ha il metodo _succ per passare allo stato successivo e _pred per passare a quello precedente. Lo stato ordinato non ha stati che lo precedono; lo stato ricevuto non ha stati che vengono dopo di esso.
- L'approccio piu` semplice sarebbe di aggiungere dei flag booleani e applicare dei semplici statement if/else all'interno di ciascun metodo. Cio` complicherebbe il nostro codice quando abbiamo piu` stati da considerare negli if/else.
- Inoltre, la logica per tutti gli stati sarebbe dissemintata tra tutti i metodi . Usiamo qundi l'approccio state-specific

- File: **esercizio_27_5_2019.py**

2

Esercizio 2

- Si consideri lo scenario in cui si ha una lista di numeri che devono essere processati in base all'intervallo a cui appartengono. Gli intervalli considerati sono [1-10], [11,20], [21,30]

- Usare lo schema nel file schemaChain.py per

1. Creare un oggetto cliente
2. Creare le richieste da processare
3. Inviare le richieste, una alla volta, agli handler come essi appaiono nella sequenza definita nella classe Client

File: **schemaChain.py**

3

Esercizio 2

- Si consideri lo scenario in cui si ha una lista di numeri che devono essere processati in base all'intervallo a cui appartengono. Gli intervalli considerati sono [1-10], [11,20], [21,30]

- Usare coroutine per implementare la stessa catena (e` possibile utilizzare Client dello schema di prima aggiungendo un'istruzione per chiudere ...).

4

Esercizio 3

- Scrivere una classe tale che ciascuna istanza della classe ha solo tre attributi: nome , cognome ed eta . Non deve essere possibile aggiungere altri attributi.
- Scrivere una classe tale che ogni istanza ha lo stesso stato

5

Esercizio 4

- Scrivere un programma concorrente (facendo prima uso di Futures e Multiprocessing e poi di joinable queue) che prende in input una lista L di stringhe ed un itero n e crea, in modo concorrente, per ciascuna delle stringhe s in L una lista. La lista creata per la stringa i-esima di L deve contenere $n//(10^i)$ occorrenze della stringa. Le liste devono essere stampate non appena vengono create.
- Ad esempio, se n=10 e L=["anna", "mario"]
- vengono stampate le liste ["anna", anna", "anna", anna", "anna", "anna", "anna", "anna", anna", "anna", anna"] ["mario"]

6

Esercizio 5

- Scrivere una classe che permetta di usare una lista in uno statement with.

7

Esercizio 6

- Scrivere una classe LaureaT_Student che puo` essere osservata e che ha i seguenti attributi che ne determinano lo stato:
- total_cfu : numero cfu acquisiti
- english_r: booleano settato a False (valore di default) se e solo se lo studente non ha superato la prova di inglese
- grades: dizionario degli esami sostenuti con elementi con chiave uguale al nome dell'esame e valore uguale al voto (exam name, grade)
- exam e` una tupla del tipo definito in basso
  - Exam=collections.namedtuple("Exam", "name cfu")
- Gli attributi total_cfu e english_r sono accessibili con il loro nome e modificabili con '=' mentre grades e` modificabile con il metodo add_grades che prende in input come primo argomento un oggetto Exam e come secondo argomento un int che rappresenta il voto

8

Esercizio 6

Scrivere inoltre i due observer HystoryView e LiveView:

- HistoryView mantiene una lista di triple della forma (dizionario degli esami sostenuti, booleano che indica se inglese superato, data cambio stato) . Ciascuna tripla e` creata quando l'oggetto LaureaT_Student cambia stato.

- LiveView esegue le seguenti stampe:

print("Cambio stato: lo studente ha appena superato la prova di Inglese\n")

se il cambio di stato e` dovuto al superamento della prova di inglese

print("Cambio stato: lo studente ha superato un nuovo esame")

print("Cambio stato: il numero di CFU e` : " , student.total_cfu,"\n")

se il cambio di stato e` dovuto al superamento di un nuovo esame

file: **observedstudents.py**

9

Esercizio 7

- Scrivere un programma in cui vi e` una classe libro che puo` essere osservata da un numero arbitrario di osservatori e che oltre all'attributo titolo ha i seguenti attributi che ne determinano lo stato:

a) riferimenti: dizionario dei riferimenti presenti al suo interno. Un riferimento e` un testo citato dal libro. Riferimenti e` quindi un dizionario di coppie (chiave, valore) dove chiave e` un intero e valore e` un libro.

b) numero_copie: un intero che rappresenta il numero di copie vendute

c) alta_progressione: flag che viene settato a True se e solo se il numero di copie aumenta almeno del doppio rispetto al valore precedente ed e` settato a False se il numero di copie aumenta meno della meta` del valore precedente (ad esempio da 10 va a 14). Se numero_copie viene aggiornato in modo diverso da quelli indicati, il flag non viene settato. Ogni volta che questo flag viene settato viene fatta la notifica agli osservatori anche se il nuovo valore del flag e` uguale a quello vecchio.

10

Esercizio 7

- Gli attributi numero_copie e alta_progressione sono accessibili con il loro nome e modificabili con ‘=‘.
- Il dizionario riferimenti viene creato e riempito dal metodo __init__ di Libro.
- Il metodo __init__ riceve in input il titolo del libro e la lista dei libri da inserire nel dizionario riferimenti. L’i-esimo libro della lista avra` chiave i nel dizionario. Attenzione: le entrate del dizionario riferimenti non devono essere modificabili con ‘=’ . Se, ad esempio, si usa l’istruzione riferimento[k] = libro allora viene lanciata l’eccezione RuntimeError.

11

Esercizio 7

- Scrivere inoltre gli osservatori VistaIst e VistaStorica.
- VistaIst deve stampare
- "Cambio stato: nuove vendite del libro \"{}\" per un totale di copie vendite pari a {} \n”, se il cambio stato e` dovuto ad un aggiornamento di numero_copie.
- "Cambio stato: con l'ultimo acquisto, il libro \"{}\" ha piu` che raddoppiato le vendite\n" se il cambio stato e` dovuto al fatto che le vendite sono raddoppiate, cioe` al fatto che alta_progressione e` stato settato a True (anche se era gia` True)
- "Cambio stato: con l'ultimo acquisto, le vendite di \"{}\" sono aumentate meno della meta`\n” se il cambio stato e` dovuto al fatto che le vendite sono aumentate meno della meta`, cioe` al fatto che alta_progressione e` stato settato a False (anche se era. gia` False) .

12

Esercizio 7

- VistaStorica crea due liste storia_vendite e andamento_vendite:
- storia_vendite e` una lista di triple della forma [titolo,numerocopie,tempo] . Ogni volta che viene aggiornato l'attributo numero_copie di uno dei libri osservati, viene aggiunta una tripla della forma [titolo,numerocopie,tempo] a storia_vendite, dove titolo e` il titolo del libro il cui numero di copie e` cambiato, numerocopie e` il nuovo numero totale di copie vendute del libro e tempo e` il tempo in cui avviene l'aggiornamento.
- andamento_vendite e` una lista di coppie della forma [stringa,tempo]. Ogni volta che viene settato alta_progressione viene inserita una coppia [stringa,tempo] in andamento_vendite, dove tempo e` il tempo in cui avviene l'aggiornamento e stringa e` "Raddoppio vendite di \"{}\"" se alta_progressione e` settato a True (anche se era gia` True) oppure e` "Incremento delle vendite di \"{}\" inferiore ad un mezzo del valore precedente" se se alta_progressione e` settato a False (anche se era gia` False).
- VistaStorica ha anche il metodo storia() che restituisce una lista il cui primo elemento e` la lista storia_vendite e il cui secondo elemento e` la lista andamento_vendite .

13

Esercizio 7

- Nelle suddette stampe al posto delle parentesi graffe devono comparire il nome del libro e/o il numero di copie, a seconda dei casi.
- Il codice relativo alla prima parte dell'esercizio deve essere scritto nel file **esercizio7_Iparte.py**. Il codice presente in esercizio7_Iparte.py non coinvolge VistaStorica. Il codice relativo alla seconda parte dell'esercizio deve essere scritto nel file **esercizio7_IIparte.py** che contiene il codice per testare VistaStorica. La classe Libro deve essere la stessa usata nella prima parte.

14

Esercizio 8

- Scrivere nel file **esercizio8.py** una funzione che prende in input una sequenza di richieste (liste di due interi) e passa ciascuna richiesta ad una catena di gestori ciascuno dei quali e` una coroutine.
- Se il primo intero della lista e` nell'intervallo [0,4] allora la richiesta viene gestita dal gestore Handler_04 che stampa "Richiesta {} gestita da Handler_04".
- Se il primo intero della lista e` nell'intervallo [5,9] allora la richiesta viene gestita da gestore Handler_59 che stampa "Richiesta {} gestita da Handler_59".
- Se il primo intero della lista e` maggiore di 9 allora la richiesta viene gestita dal gestore Handler_gt9 che stampa "Messaggio da Handler_gt9: non e` stato possibile gestire la richiesta {}. Richiesta modificata". Dopo aver effettuato la stampa Handler_gt9 sottrae al primo intero della lista il secondo intero della lista e lo invia nuovamente ad una nuova catena di gestori.
- Se la richiesta non è una lista di due numeri o il primo intero della lista e` minore di 0 la richiesta viene gestita da Default_Handler che stampa semplicemente "Richiesta {} gestita da Default_Handler: non e` stato possibile gestire la richiesta {}".
- Nelle suddette stampe la lista nella richiesta deve comparire al posto delle parentesi graffe.

15

Esercizio 9

- Scrivere nel file **esercizi9_da_6.py** o nel file **esercizio9_da_10.py**, a seconda della versione che si decide di svolgere, una coroutine searcher(c1,c2,receiver1, receiver2) che prende in input due caratteri c1 e c2 e due coroutine receiver1, receiver2 , e si comporta come segue: ogni volta che riceve qualcosa verifica se questa e` il nome di un file esistente e nel caso in cui lo sia cerca all'interno del file le stringhe che cominciano con c1 e quelle che cominciano con c2.
  - Le prime vengono inviate a receiver1 mentre le seconde a receiver2. Nel caso in cui non esista un file con quel nome, la coroutine esegue solo la stampa della seguente stringa "Il file {} e` inesistente", dove al posto delle parentesi deve comparire il nome del file.
- Scrivere inoltre una coroutine listCreator(stop) che ogni volta che riceve una stringa la inserisce in una lista (la lista e` una variabile locale alla coroutine) e stampa la lista aggiornata con l'aggiunta della nuova parola. I parametri receiver1 e receiver2 di searcher sono due coroutine listCreator.
- Versione da al massimo 6 punti: la coroutine listCreator non fa niente altro rispetto a quanto sopra descritto (l'input stop viene ignorato).

16

Esercizio 9

- Versione da al massimo 10 punti: La couroutine smette di ricevere parole non appena riceve una parola uguale alla stringa stop passata come argomento. Nell'implementazione della coroutine searcher occorre tenere conto del fatto che uno o entrambi i receiver potrebbero non ricevere piu` le parole inviate. Se ad un certo punto entrambi i receiver smettono di ricevere parole il searcher deve smettere anch'esso di ricevere stringhe.
- Suggerimento: potete usare re.findall(r'\w+', testo) per estrarre parole da un testo.

17