Gestione File

Mirko Cesarini - Dario Pescini
nome.cognome@unimib.it

Università di Milano Bicocca

Hardware di un computer

- **Memoria centrale** (o RAM)

- Caratteristiche
  - **Memoria principale di lavoro**
  - Accesso molto veloce ai dati
  - Capienza limitata
  - Volatile (spegnendo il computer, si perdono tutte le informazioni)

- **Memoria di massa** (disco rigido, . . .)

- Caratteristiche
  - **Memoria secondaria di lavoro**
  - Capienza molto maggiore (rispetto alla memoria centrale)
  - Accesso più lento ai dati (rispetto alla memoria centrale)
  - Non volatile (spegnendo il computer, NON si perdono le informazioni)

Paragone con le biblioteche

- Scenario: una persona cerca dei libri negli scaffali, poi li porta con se su una scrivania e inizia a consultarli
- Sulla scrivania
  - i libri presenti possono essere consultati velocemente
  - è possibile appoggiare solo un numero limitato di libri
  - se è necessario un libro non presente nella scrivania, occorre andarlo a cercare negli scaffali
  - se, dopo aver cercato nuovi libri, nella scrivania non c'è più spazio, occorre riporre qualcosa negli scaffali
- Negli scaffali
  - capienza molto maggiore della scrivania
  - tuttavia recuperare un libro richiede tempo

Analogia tra computer e biblioteca

- Memoria centrale. Equivalente alla scrivania della biblioteca
  - Variabili python: ospitate nella memoria centrale.
  - Le variabili ereditano vantaggi e svantaggi della memoria centrale
    - Velocità
    - Capienza limitata
    - Volatilità (al mancare della corrente si perde tutto)
- Memoria di massa. I dati sono organizzati sotto forma di file e directory.
  - File: contenitore di dati (equivalente al libro)
  - Directory: contenitore di file (equivalente ad uno scaffale della biblioteca)

File: motivazioni

- Se devo spegnere il computer, come faccio a non perdere i risultati ottenuti?
    - Salvo i dati nella memoria di massa
- Se devo processare una grossa quantità di informazioni le cui dimensioni eccedono la capienza della memoria centrale di un computer?
    - memorizzo i dati nella memoria di massa
    - carico in memoria centrale ed elaboro un sottoinsieme di dati alla volta
- Obiettivo della lezione: vedere la gestione dei file in Python

Altro paragone

- "Lavorare con i file è simile a leggere un libro: per usarli li devi prima **aprire** e quando hai finito li **chiudi**. Mentre il libro è aperto puoi **leggerlo**, puoi **scrivere** una nota sulle sue pagine. La maggior parte delle volte leggerai il libro in ordine, una pagina dopo l'altra, ma nulla ti vieta di **saltare** a determinate pagine facendo uso dell'indice."
- **Apertura**, **chiusura**, **lettura**, **scrittura** e **posizionamento** sono le attività tipiche che un linguaggio di programmazione mette a disposizione per gestire il contenuto di un file ...
- ... i linguaggi di programmazione gestiscono ulteriori attività: la creazione di un nuovo file e la cancellazione del contenuto di un file esistente

Due operazioni frequenti: apertura e chiusura

- **Apertura** di un file: si informa il sistema operativo . . .
  - . . . che si vuole utilizzare un certo file. Il sistema operativo effettua dei controlli prima di consentire l'accesso (Il file esiste? L'utente dispone dei permessi di accesso? Qualche altro utente lo sta utilizzando?).
  - . . . indicando quali operazioni saranno svolte (solo lettura, scrittura, . . .)
- **Chiusura** di un file: si informa il sistema operativo che il programma non ha più bisogno di leggere e/o scrivere sul file
  - (per ottimizzare le prestazioni) le operazioni di lettura e scrittura dei dati non vengono eseguite appena richieste
  - Quando un file viene chiuso, il sistema operativo cerca di completare velocemente le operazioni in sospeso
  - Domanda: perché è necessario informare il sistema operativo prima di staccare una chiavetta usb?
  - Quando si informa il sistema operativo che si vuole staccare un dispositivo di memorizzazione esterno, il sistema operativo completa le operazioni in sospeso prima di dare l'ok alla rimozione

Python: apertura e chiusura di un file

- Esempio di apertura di due file

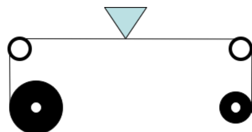
```
f1 = open("source.dat","r")
f2 = open("destination.dat","w")
```

- La funzione *open* accetta due argomenti:
  - il nome del file
  - la modalità di apertura (maggiori dettagli nelle slide successive)
    - (Sola) lettura: "r"
    - (Lettura e) scrittura: "w"
    - Append: "a"
    - ...
- *f1* ed *f2* sono i descrittori dei file
  - Descrittore: una variabile che rappresenta il file nello script
  - I comandi che implementano le operazioni sul file saranno richiamati con la *notazione punto* sul descrittore es:,
    - `f1.close()  #sara' descritto in seguito`
    - `f2.read()   #sara' descritto in seguito`
- A breve torneremo sul descrittore del file e sulla *notazione punto*

Python: modalità di apertura di un file

- "r". Chiamata anche *read*, o *(sola) lettura*: vogliamo aprire un file solo per leggerne il contenuto (senza scriverci sopra). Alcuni sistemi operativi permettono a più utenti contemporaneamente di aprire un file in lettura. Se si tenta di scrivere su un file aperto in sola lettura, si genera un errore e il programma termina.
- "w". Chiamata anche *write*, o *scrittura*, si desidera aprire un file per poterci scrivere sopra
  - `f2 = open("destination.dat","w")`
    Se non esiste già un file chiamato *destination.dat* l'apertura in scrittura creerà un nuovo file (vuoto)
  - Se il file esiste già, il contenuto preesistente sarà eliminato
- "a". *append*: simile alla modalità *write*, ma il contenuto preesistente del file non viene cancellato, i nuovi dati saranno accodati ai dati già esistenti. Se il file è vuoto o non esiste già, non c'è differenza tra modalità "w" e "a"
- ...esistono altre modalità di apertura che non vedremo

Modello del *nastro magnetico*



- Una testina di lettura / scrittura
- La testina scorre su un "nastro" logicamente suddiviso in caselle (una di seguito all'altra)
  - ogni casella rappresenta un byte
  - le caselle sono numerate ...
    - a partire dalla posizione 0
    - fino alla fine del file
- i file sono memorizzati uno dopo l'altro (come più film su una videocassetta)
- Operazioni di lettura/scrittura: il nastro scorre, la testina legge/scrive caselle contigue
- Dopo l'ultima operazione di lettura/scrittura, la testina è **posizionata** all'inizio della casella successiva (pronta per una nuova operazione)
- Il nastro può essere fatto scorrere velocemente, senza dover leggere quello che c'è in mezzo

File: note sul posizionamento

- Quando si apre un file in modalità sola lettura ("r")
  - la testina viene posizionata all'inizio del file
  - ogni volta che si leggono dei dati il nastro scorre
  - lettura dopo lettura, la testina arriva fino alla fine del file
  - è possibile effettuare dei salti, sia avanti, sia indietro
- Quando si apre un file in modalità scrittura ("w")
  - viene cancellato tutto il contenuto del file esistente
  - la testina viene posizionata all'inizio del file pronta per scrivere
  - dopo ogni scrittura, la testina si posiziona alla fine dell'ultima cella letta/scritta (pronta per eseguire l'operazione successiva)
  - è possibile effettuare dei salti, sia avanti sia indietro
- Quando si apre un file in modalità "append" ("a")
  - la testina viene posizionata alla fine del file
  - i nuovi contenuti vengono aggiunti (scritti) in coda
  - dopo ogni scrittura, la testina si posiziona alla fine dell'ultima cella letta/scritta (pronta per eseguire l'operazione successiva)

Python: esempi di lettura e scrittura

- Apertura di un file

```
1 f = open("test.txt","w")
```

- Per scrivere dati nel file occorre utilizzare il metodo write:

```
2 f.write("Adesso")
3 f.write("Scriviamo qualcosa")
```

- Modello nastro magnetico: i due comandi write scrivono le 2 frasi una di seguito all'altra
- Notazione punto dei comandi es., `f.write()` (sarà ripresa in seguito)
- La chiusura del file avvisa il sistema operativo che la scrittura è conclusa ed il file è disponibile per altri scopi:

```
4 f.close()
```

- Solo dopo aver chiuso il file possiamo riaprirlo in lettura e leggerne il contenuto
- Se cerchiamo di aprire in lettura un file che non esiste otteniamo un errore:

```
1 h = open("test.cat","r")
```

IOError: [Errno 2] No such file or directory: 'test.cat'

- Nel comando open possiamo
  - fornire solo il nome del file
    in tal caso il file sarà ricercato nella directory corrente
  - fornire il percorso completo, es.

```
2 k = open("c:\\notes\\lecture1.txt","r")
```

- Nelle stringhe python, quando si vuole inserire il \ semplice, questo va scritto raddoppiato
- Perchè secondo voi?
- Per evitare che il \ e i caratteri successivi vengano interpretati come comandi di formattazione. In questo modo il \n presente nella stringa non viene interpretato come *a capo*

Lettura

- Reminder

```
f = open("test.txt","w")
f.write("Adesso")
f.write("Scriviamo qualcosa")
f.close()
```

- Immaginiamo di riaprire il file dopo un po' di tempo
- Questa volta la modalità di apertura è "r":

```
g = open("test.txt","r")
```

- Il metodo read legge dati da un file. Senza argomenti legge l'intero contenuto del file:

```
testo = g.read()
print( testo )
```

```
AdessoScriviamo qualcosa
```

- Notate qualcosa di strano?
- Non c'è spazio tra *Adesso* e *Scriviamo* perché la seconda stringa è stata scritta subito dopo la prima
- read accetta anche un argomento che specifica quanti caratteri leggere

```
g = open("test.txt","r")
print(g.read(5))
```

```
Adess
```

```python
f = open("caratteri.txt","w")
f.write("abcdefgh")
f.close()
```

```python
g = open("caratteri.txt","r")
txt = g.read(5)
print(txt) # abcde
```

- In un'operazione di lettura, read restituisce solamente i caratteri effettivamente disponibili,
- Es: si richiede di leggere 5 caratteri, ma il file termina dopo averne letti 3. In qs caso read restituisce solamente i caratteri effettivamente letti

```python
txt2 = g.read(5)
print(txt2) # fgh
```

- Se si cerca di leggere qualcosa dopo aver raggiunto la fine del file, read restituisce una stringa vuota

```python
txt3 = g.read()
print(len(txt3)) # 0
```

Copiare il contenuto di un file su un altro file

- Esercizio su lettura e scrittura di un file

```
f1 = open("source.dat","r")
f2 = open("destination.dat","w")
print('Copying content')
content = f1.read()
f2.write(content)
f1.close()
f2.close()
print('Done')
```

- Questo script presuppone che l'elaboratore abbia abbastanza memoria per poter caricare in memoria centrale l'intero contenuto del file

Variabili, Oggetti, Metodi

- Abbiamo visto che sulle variabili che ospitano i descrittori di file è possibile eseguire delle operazioni con la *notazione punto*
- Le operazioni richiamate con la *notazione punto* sono chiamate *metodi*
- *read*, *write* e *close* sono metodi messi a disposizione da una variabile che ospita un descrittore di file
- Abbiamo già visto altri esempi di metodi

```
1 a=[]
2 a.append(5)
3 print(a) # [5]
```

- Una variabile su cui possono essere richiamati dei metodi è chiamata *oggetto*
- In python tutte le variabili sono oggetti. Questo significa che ogni variabile mette a disposizione dei metodi

- I metodi richiamabili dipendono dal tipo di valore ospitato dalla variabile

```
a=[]
a.append(5)
print(a) # [5]
f=open('abc.txt', 'r')
cont=f.read()
print(cont) # ...
f.close()
```

ok

```
f=open('abc.txt', 'r')
f.append('5')
```

AttributeError: 'file' object has no attribute 'append'

```
a=[]
a.close()
```

AttributeError: 'list' object has no attribute 'close'

Sintassi alternativa: with

Fino ad ora abbiamo visto questa sintassi per aprire, manipolare e chiudere un file

```
f1 = open('welcome.txt', 'r')
data = f1.read()
print(data)
f1.close()  # E' importante chiudere il file
            # quando non lo si usa piu'
print('Operazione terminata e file chiuso')
```

- Python mette a disposizione una sintassi alternativa

```
with open("welcome.txt") as f2:
    data = f2.read()
    print(data)
print('Operazione terminata e file chiuso')
```

- Secondo voi, nella seconda variante ci siamo dimenticati di chiudere il file?
- No. Lo fa automaticamente l'interprete Python alla fine del blocco indentato

File di testo

- Un file di testo è un file che contiene caratteri stampabili e spazi bianchi, organizzati in linee separate da caratteri di ritorno a capo.
- Per esempio il testo seguente,

```
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.
```

- memorizzato in un file, apparirebbe in questo modo:
  Nel_mezzo_del_cammin_di_nostra_vita\nmi_ritrovai_per
  _una_selva_oscura,\nché_la_diritta_via_era_smarrita.\n
- Nota: \n significa *a capo*, _ significa *spazio*

Carattere di a capo

Il carattere di a capo può essere rappresentato da:

- '\n' chiamato Line Feed (o New Line)
- '\r' chiamato Carriage Return
- Una combinazione dei due precedenti, es. il notepad di windows usa '\r\n'
  - se ne manca uno, il notepad non riconosce l'*a capo*
  - per questo motivo il notepad a volte non visualizza il contenuto dei file di testo correttamente

Python: I/O di file di testo

- Nel nostro corso ci occuperemo principalmente di file di testo
- Creazione di un file di testo composto da tre righe separate da caratteri di *a capo*:

```
1  f = open("test.txt","w")
2  f.write("linea uno\nlinea due\nlinea tre\n")
3  f.close()
```

- Il metodo readline legge tutti i caratteri, dalla posizione corrente fino al prossimo ritorno a capo:

```
1 f = open("test.txt","r")
2 print ( f.readline() )
```

```
linea uno
```

Fine file

- Quando il file è stato letto tutto, readline restituisce una stringa vuota:

```
print ( f.readline() )
```

← La stringa vuota

- Si può utilizzare quanto appena detto come test per verificare se ci sono ancora righe da leggere

```
c = open("test.txt","r")
linea = c.readline()
while linea!="": # "" niente spazi in mezzo
    print (linea)

    linea = c.readline()
```

- Domanda: è possibile confondere una riga bianca/vuota con la fine del file?
- No. Ogni riga contiene almeno un carattere. Es., una riga bianca contiene almeno un \n (new line)

Scrittura

- L'argomento della write dev'essere una stringa, se vogliamo memorizzare altri tipi di valore in un file di testo occorre convertirli in stringhe.
- Il modo più semplice è quello di usare la funzione str():

```
1 x = 52
2 f.write(str(x))
```

- Un esempio più complesso di conversione in stringa

```
3 NumAuto=100
4 a="In luglio abbiamo venduto %d automobili." % NumAuto
5 print(a)
```

```
In luglio abbiamo venduto 100 automobili
```

```
6 f.write(a)
```

Posizionamento all'interno del file

- I dati vengono letti/scritti a partire dalla posizione corrente del file (la testina nel modello precedentemente introdotto)
- E' possibile variare il punto in cui verranno letti/scritti i prossimi dati con il metodo seek

```
f = open ("dati.dat", "r")
f.seek(15) # la posizione corrente diventa la 16ma
           # casella (sono numerate a partire da 0)
```

- Altra funzione utile: f. tell () restituisce l'indice numerico corrispondente alla posizione corrente all'interno del file
  - pos = f. tell () memorizza in una variabile la posizione corrente del file

Esempi di lettura file

- Ciclo che legge tutte le righe:

```
f = open("prova.dat", "r")
line=f.readline()
while line!="":
    print (line)
    line=f.readline()
```

- Il ciclo for permette di riscrivere il ciclo while di cui sopra in forma sintetica

```
f = open("dati.dat", "r")
for line in f:
   print(line)
```

- Il ciclo while permette di variare l'ordine di lettura, nel ciclo for non è consigliabile farlo

Alcuni trucchi utili

- Il carattere \n (new line) a volte da fastidio

```
f = open("test.txt","w")
f.write("linea uno\nlinea due\nlinea tre\n")
f.close()
c = open("test.txt","r")
linea = c.readline()
if linea == "linea uno":
    print("E' uguale')
 else:
    print("E' diverso")
```

- Cosa viene stampato?
- Viene stampato *E' diverso*. Motivo: "linea uno"!="linea uno\n"
- Il metodo strip permette di risolvere il problema

```
linea="linea uno\n" # strip elimina il carattere passato
v=linea.strip("\n") # come parametro (se presente) al-
print (v)           # l'inizio e alla fine della stringa
```

```
linea uno
```

Trucchi 2: split

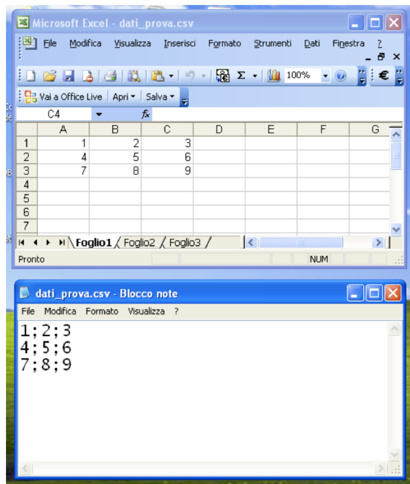
- Spesso si ha a che fare con file CSV (Comma separated value)
- Esempio del contenuto di un file CSV
  Paolo,Rossi,25
  Gaetano,Scirea,28
  ...
- Leggendo, una riga, il metodo **split** permette di ottenere una lista in cui gli elementi sono ...

```
1 c = open("prova.csv","r")
2 linea = c.readline()
3 valori = linea.split(",")
4 print(valori)
```

```
['Paolo', 'Rossi', '25\n']
```

File CSV – MS Excel

- I file CSV sono spesso usati per scambiare dati tra applicazioni
- MS Excel può sia leggere sia salvare i dati di un foglio di calcolo in formato .CSV
  - Fate una prova: aprite excel, inserite dei dati in alcune celle e salvate il file in formato CSV
  - Aprite il file con un editor di testi
- I file CSV quindi non sono altro che file di testo



Esercizio 1

Dato il file persone.csv

```
Nome;Cognome;eta\n
Mario;Rossi;10\n
Giuseppe;Verdi;20\n
...
```

Lo script seguente permette di leggere i dati

```python
f=open('persone.csv','r')
prima=True
for line in f:
    if prima==True: # Secondo voi, a cosa serve questa?
        prima=False
    else:
        line = line.strip('\n')
        dati=line.split(';')
        (nome, cognome, eta)=dati
        print(nome, cognome, eta)
f.close()
```

Risposta: cosi non viene processata la prima riga

Esercizio 1 - Variante A

Stesso dataset, ma visualizziamo solo la media delle età

```
Nome;Cognome;eta\n
Mario;Rossi;10\n
Giuseppe;Verdi;20\n
...
```

```
1  f=open('persone.csv','r')
2  f.readline() # sposto la testina di lettura
3               # all'inizio della seconda riga
4  somma=0
5  n=0
6  for line in f: # ora parte dalla seconda riga
7      line = line.strip('\n')
8      dati=line.split(';')
9      eta=dati[2]
10     eta=int(eta) # converto da stringa a int
11     somma+=eta
12     n+=1
13 f.close()
14 print(somma/float(n))
```

Esercizio 1 - Variante B

Stesso dataset, script simile al precedente.

```
Nome;Cognome;eta\n
Mario;Rossi;10\n
Giuseppe;Verdi;20\n
...
```

```
1  f=open('persone.csv','r')
2  f.readline()
3  somma=0
4  n=0
5  for line in f:
6      line = line.strip('\n')          # Se eliminiamo qs riga,
7      dati=line.split(';')            # creiamo problemi?
8      eta=dati[2]
9      eta=int(eta)
10     somma+=eta
11     n+=1
12 f.close()
13 print(somma/float(n))
```

(La riga 6, `line = line.strip('\n')`, è barrata.)

No. Il '\n' non causa problemi a `int(eta)`, viene ignorato.

Esercizio 2

Dato il file iscrizioni.csv

```
Corso;Matricola_studente_iscritto\n
Informatica;75731\n
informatica;69489\n
analisi;84893\n
Analisi;74176
```

Creo un dizionario che conta il numero di studenti iscritti per corso

```python
fin=open('iscirizioni.csv','r')
content=fin.read() # leggo tutto il file
lines=content.split('\n') # spezzo sulla base dei \n
# le 2 istruzioni qua sopra potrebbero essere sostituite da
#lines=fin.readlines()

lines=lines[1:] # elimino il 1. elemento della lista che
                # corrisponde alla riga di intestazione
# ... continua

# Script completo nella slide successiva
```

Esercizio 2 - Script completo

```
fin=open('iscirizioni.csv','r')
content=fin.read() # leggo tutto il file
lines=content.split('\n') # spezzo sulla base dei \n
# le 2 istruzioni qua sopra potrebbero essere sostituite da
#lines=fin.readlines()

lines=lines[1:] # elimino il 1. elemento della lista che
                # corrisponde alla riga di intestazione

conteggi={}
for row in lines:
    data = row.split(';')
    corso=data[0]
    corso=corso.upper() #converte in lettere maiuscole
    if corso in conteggi:
        conteggi[corso]+=1
    else:
        conteggi[corso]=1
print(conteggi)
```

Tema d'esame dell'8/mag/2017

- Il file telefonate.csv contiene informazioni sulle telefonate effettuate in un *unico* mese da alcuni clienti di un operatore di telefonia cellulare. La prima riga contiene l'intestazione delle colonne.
  Cod_Chiamante;Cod_Destinatario;Cod_Cella_Chiamante;
  Cod_Cella_Destinatario;GGI:HHI:MMI;GGF:HHF:MMF\r\n
  - Cod_Chiamante: un intero che identifica univocamente il cliente che ha effettuato la chiamata
  - Cod_Destinatario: un intero che identifica univocamente il cliente che ha ricevuto la chiamata
  - Cod_Cella_Chiamante: un intero che identifica l'area nella quale si trova il cliente che ha effettuato la chiamata
  - Cod_Cella_Destinatario: un intero che identifica l'area nella quale si trova il cliente che ha ricevuto la chiamata
  - GGI:HHI:MMI rappresentano rispettivamente il giorno, l'ora e il minuto in cui e' iniziata la chiamata
  - GGF:HHF:MMF rappresentano rispettivamente il giorno, l'ora e il minuto in cui e' finita la chiamata
- Le righe che iniziano con # sono da scartare
- Il \r\n rappresentano il simbolo di "a capo".

Specifica dei requisiti - ottieniDatiTelefonate()

Implementate la funzione ottieniDatiTelefonate (nomeFile) che accetta in ingresso il nome del file da elaborare e restituisce una lista di tuple, dove ogni tupla corrisponde ad una telefonata e contiene

(Cod_Chiamante, Cod_Destinatario, Cod_Cella_Chiamante, Cod_Cella_Destinatario, Numeo_Minuti_Di_Conversazione)

I minuti di conversazione si calcolano includendo anche il minuto di inizio della conversazione

Devono essere escluse le righe che rispettano i seguenti criteri:

- Cod_Chiamante e Cod_Destinatario devono essere diversi
- GG, HH e MM devo appartenere ai rispettivi domini es $HH \in 0 \ldots 23$
- L'inizio della chiamata non puo' essere successivo alla fine
- Telefonate piu' lunghe di 10 ore sono errori e non entrano nel risultato restituito

Per il calcolo dei minuti di conversazione, si suggerisce di trasformare (data,ora,minuto) in un intero corrispondente al numero di minuti trascorsi dall'inizio del mese. I minuti di conversazione si calcolano includendo anche il minuto di inizio della conversazione ($+= 1$ sui minuti di conversazione)

Implementazione - funzioni accessorie

```
def check(gg, hh, mm):
    if gg < 1 or gg > 31:
        return False
    if hh < 0 or hh > 23:
        return False
    if mm < 0 or mm > 59:
        return False
    return True  # controlli ok
```

```
def tempoTrascorso(gg, hh, mm):
    return mm + hh * 60 + gg * 24 * 60
```

Implementazione - ottieniDatiTelefonate()

```
11 def ottieniDatiTelefonate(nomeFile):
12     # (Cod_Chiamante, Cod_Destinatario, Cod_Cella_Chiamante,
13     # Cod_Cella_Destinatario, Numeo_Minuti_Di_Conversazione)
14     data = []
15     fi = open(nomeFile, 'r')
16     fi.readline()  # salto la 1. riga
17     for line in fi:
18         if line[0]!='#':
19             line = line.strip('\r\n')
20             line = line.split(';')
21             chiamante = int(line[0])
22             destinatario = int(line[1])
23             cellaChiamante = int(line[2])
24             cellaDestinatario = int(line[3])
25             timeInizio=line[4]
26             timeFine=line[5]
27 
28             timeInizioLi=timeInizio.split(':')
29             timeFineLi=timeFine.split(':')
30             # ... continua
```

```
31            # ...
32            ggi = int(timeInizioLi[0])
33            hhi = int(timeInizioLi[1])
34            mmi = int(timeInizioLi[2])
35            ggf = int(timeFineLi[0])
36            hhf = int(timeFineLi[1])
37            mmf = int(timeFineLi[2])
38
39            inizio = tempoTrascorso(ggi, hhi, mmi)
40            fine = tempoTrascorso(ggf, hhf, mmf)
41            delta = fine - inizio
42            if chiamante != destinatario and
43                    check(ggi, hhi, mmi) and
44                    check(ggf, hhf, mmf) and delta >= 0
45                    and delta <= 60*10:
46                durata = delta + 1  # conteggio 1. minuto
47                el = ( chiamante, destinatario,
48                       cellaChiamante, cellaDestinatario,
49                       durata )
50                data.append(el)
51        fi.close()
52        return data
```

Specifica dei requisiti - calcolaBollette()

Implementa la funzione `calcolaBollette (datiChiamate)`. Parametro in ingresso la struttura dati restituita dalla funzione precedente.
Deve restituire un dizionario di coppie chiave valore codice_cliente:importo dove il valore e' un float con l'importo che il cliente deve pagare per le chiamate effettuate nel mese.
Un cliente paga 0.10 euro per ogni minuto di chiamata effettuata. Il cliente che riceve la chiamata non paga niente. Ogni cliente ha un bonus di 20 minuti gratuiti al mese
Per esempio se il cliente 1 effettua chiamate per 55 minuti e il cliente 2 effettua chiamate per 15 minuti, il dizionario restituito dovra' contenere: `{1:3.5, 2:0}`.
L'insieme dei clienti contenuti nel dizionario e' formato da quei clienti che appaiono almeno una volta come chiamanti nella struttura dati fornita in ingresso.

Implementazione - calcolaBollette()

```
def calcolaBollette(datiChiamate):
    # [chiamante,destinatario, cellaChiamante,
    #  cellaDestinatario, durata]
    clienti2minuti = {}
    for el in datiChiamate:
        chiamante = el[0]
        durata = el[4]
        if chiamante in clienti2minuti:
            clienti2minuti[chiamante] += durata
        else:
            clienti2minuti[chiamante] = durata
    clienti2tariffa = {}
    for cliente in clienti2minuti:
        minuti = clienti2minuti[cliente]
        if minuti <= 20:
            clienti2tariffa[cliente] = 0
        else:
            clienti2tariffa[cliente] = (minuti - 20) * 0.10
    return clienti2tariffa
```

Specifica dei requisiti - calcolaCelleCongestionate()

Implementate la funzione calcolaCelleCongestionate (datiChiamate).
La funzione accetta come parametro in ingresso la struttura dati creata dalla funzione ottieniDatiTelefonate.
La funzione deve identificare le celle (cioe' le aree) che hanno un elevato carico di telefonate.
Per far cio', dovete calcolare il numero totale di telefonate in partenza effettuate per cella.
Sono da considerarsi congestionate le celle nelle quali il numero di telefonate effettuate e' maggiore della media (calcolata sui totali precedentemente calcolati).
La funzione dovra' restituire una lista con i codici delle celle congestionate.

Implementazione - calcolaCelleCongestionate()

```
1  def calcolaCelleCongestionate(datiChiamate):
2      # (chiamante,destinatario, cellaChiamante,
3      #  cellaDestinatario, durata)
4      celle2chiamate = {}
5      for el in datiChiamate:
6          cella = el[2]
7          if cella in celle2chiamate:
8              celle2chiamate[cella] += 1
9          else:
10             celle2chiamate[cella] = 1
11
12     numChiamate = 0
13     numCelle = 0
14     # ... continua
```

```
14      # ...
15      for cella in celle2chiamate:
16          n = celle2chiamate[cella]
17          numChiamate += n
18          numCelle += 1
19      media = float(numChiamate) / numCelle
20      congestionate = []
21      for cella in celle2chiamate:
22          n = celle2chiamate[cella]
23          if n > media:
24              congestionate.append(cella)
25      return congestionate
```