Dbase un database relazionale in Python con interfaccia ad oggetti.

Gestito completamente in memoria, sfrutta file json come archivio.

Database monoutente liberamente ispirato al DB III+

Marco Salvati

Cos'è Dbase

Dbase è un mini framework in Python per creare database locali, con interfaccia ad oggetti, utilizza un file json come database, è gestito interamente in memoria.

Liberamente ispirato al dbase III+, il database relazionale monoutente leader negli anni 80 e 90, ricrea la sua filosofia di gestione degli archivi applicata a dei file json

Il database è uno o più file di testo apribili con un semplice editor, caricato in memoria questi diventa a tutti gli effetti un database relazionale, dbase permette molte operazioni like dbase III+.

Anche se i file sono in json non ci sono limitazioni su tipi, Dbase converte in automatico i tipi non riconosciuti da questo formato ed è facilmente espandibile, per tipi non previsti.

Questo tutorial guiderà passo dopo passo al suo utilizzo.

Fasi di sviluppo del progetto:

- Creazione delle primitive modulo dbs.py
- Una classe singleton per la gestioni di oggetti dbs modulo Shared.py
- Gestione delle date modulo Date.py
- Gestione dei campi del database modulo field_type.py
- Gestione delle tabelle modulo Table.py
- Gestione di gruppi di campi modulo Record.py
- Indicizzazione del database modulo Idbs.py
- Relazione tra campi indici di database diversi tramite una chiave univoca, modulo dbRelaction.py
- Funzioni matematiche su database modulo dbMath.py

Modulo dbs.py:

Questa classe è il core del database non viene mai utizzata direttamente se non per eventuali estensioni, contiene le primitive per la creazione e gestione del database.
Sintassi: dbObj=dbs(filename)

free_field()	**Ritorna il prossimo indice a cui associare il campo in fase di dichiarazione**
seek(record)	**Posiziona il puntatore sul record desiderato**
seek_to_start()	**Posiziona il puntatore sul primo record**
seek_to_end()	**Posiziona il puntatore sull'ultimo record**
tell()	**Ritorna la posizione del puntatore del file**
eof()	**Crea un record vuoto**
fieldType(field)	**Ritorna il tipo di campo**
fieldValue(record,field)	**Ritorna il valore di un campo**
use(name="")	**Carica il database, se name non è in presente usa il Tablename con L'estensione '.tab' nella directory corrente ritorna 0 ok, -1 il file non valido, 2 non esiste**
flush(name="")	**Salva il database, può utilizzare un nome diverso**
dbCreate(name="")	**Crea la tabella sul disco , se filename non è in presente usa il Tablename con L'estensione '.tab' nella directory corrente**
isValidDb(name):	**Controlla se è un database valido, ritorna True se ok**
dbAccess(filename)	**Ritorna true se il database o qualsiasi altro file esiste**
__getitem__(key)	**Magic method legge una chiave dal record attuale**
__setitem__(key,value)	**Magic method setta una chiave nel record attuale**
__delitem__(key)	**Elimina una record key=0 record attuale è permesso lo slice per cancellazione di record contigui multipla (va usata con cautela)**
__next__()	**Magic method prossimo record next(dbsObj)**
__len()__	**Ritorna il numero di record del database**
sort(field,reverse=False)	**Sort del database per il campo assegnato**
dbGet()	**Ritorna una copia dell' intero database**
search(dato,field)	**Ricerca sequenziale**
next()	**Continua la ricerca iniziata da search**

Modulo Shared.py

Si tratta di una class singleton crea e condivide i vari oggetti dbs, anche questa importante classe non viene utilizzata mai direttamente se non per costruire eventuali estensioni.

set_table(table)	Crea un oggetto dbs e lo associa ad una tabella
get_table(table=None)	Ritorna il dbs associato ad una tabella, della tabella attuale se None
set_use_table(table)	Setta la tabella in uso
get_use_table()	ritorna la tabella in uso

Questa classe implementa i magic method metodi per dizionari:
__ getitem__ __setitem__ __delitem__ __contains__ __len__

E i metodi **keys, values, items** inseriti per future estensioni.

Modulo Date.py

Ora vediamo la classe Date che ci permettera la gestione delle date nel database e non solo.

Sintassi: dt=Date(anno,mese,giorno) Crea una nuova data

Metodi:

dmy() Ritorna la data in formato gg-mm-aaaa

mdy() Ritorna la data in formato mm-gg-aaaa

ymd() Ritorna la data in formato aaaa-mm-gg

year() Ritorna l'anno della data

month() Ritorna il mese della data

day() Ritorna il giorno della data

today() Ritorna la data attuale *(classmethod)*

age() Ritorna quanti anni, mesi, settimane e giorni sono trascorsi o mancano rispetto la data attuale.

day_split(giorni) Trasforma i giorni in una tupla (anni, mesi, settimane e giorni_restanti)

split(data) Crea una nuova oggetto Date da una stringa in formato aaaa-mm-gg

add_days(giorni) Aggiunge alla data un certo numero di giorni

sub_days(giorni) Sottrae alla data un certo numero di giorni

isleap(year) Ritorna True se l'anno è bisestile

Magic method:

__sub__(otherDate) Ritorna il numero di giorni tra due date gg=data1-data2

__str__ __repr__ Ritorna la data come stringa e in formato evalutabile

__call__(anno, mese, giorno) Cambia la data in un oggetto Date

__qe__ __qt__ __eq__ __le__ __lt__ __ne__ confronto tra due oggetti data data1 > data2 etc..

Ora siamo pronti per i moduli di interfacciamento al modulo dbs

Modulo Field_type.py

Field_type è una classe astratta da cui derivano tutte le classi per la gestione dei campi permette di creare facilmente nuovi tipi di dati .

value Ritorna o setta il valore del campo

Magic method:

__eq__ __qe__ __qt__ __le__ __lt__ __nt__ confronto

È possibile confrontare un campo con un valore senza accedere al suo valore.

Esempio:
if impiegato.Cognome==’Rossi’ invece di if impiegato.Cognome.value==’Rossi:

Per i numeri complessi sono stati implementati tutti i metodi di confronto la classe complex standart ammette solo != , quindi sara possibile un confronto tipo
if tab.Complesso >= 3+7j: non implementato in complex.

__str__ Ritorna come stringa

__repr__ Ritorna una stringa evalutabile

Attualmente ho previsto 13 tipi di dato

- ***Field_int(Range=(min, max))*** ***Campo interi con e senza range***
- ***Field_float(Range=(min, max))*** ***Campo float con e senza range***
- ***Field_complex(Range=(min, max))*** ***Campo complex con e senza range***
- ***Field_string(size)*** ***Campo stringa di size caratteri***
- ***Field_char(valid_char=None)*** ***Campo di un carattere, valid_char se presente è una tupla di due elementi una stringa di caratteri validi e il cattere di default***
- ***Field_text()*** ***Campo Testo senza limiti di dimensione (ricorda i campi memo del dbIII+)***
- ***Field_bool()*** ***Campo booleano***
- ***Field_tuple()*** ***Campo tuple***
- ***Field_dict()*** ***Campo dizionari***
- ***Field_set()*** ***Campo set***
- ***Field_list()*** ***Campo liste***
- ***Field_date()*** ***Campo date***
- ***Field_setof(valid)*** ***Campo valori escusivi accetta solo uno dei valori in valid***

Esempio di dichiarazioni delle classi Field:

Intero1=Field_int() *Definisce un campo intero Python 3 senza limiti*

Byte=Field_int(Range=(-127,128) *tipo range int*

Float=Field_float() *Definisce un campo float*

Float=Field_float(Range)=(10.5,30.7) *Campo range float*

Bool=Field_bool() *Campo di tipo bool*

Complex=Field_complex() *Definisce un campo di tipo complex*

Complex2=Field_complex(Range=(2+3j,7+0j)) *complex con range*

String= Field_string(50) *Campo stringa di 50 caratteri*

Char=Field_Char() *Campo di un solo carattere*

Char2=Field_Char("MF", "F") *Campo di un solo carattere solo M o F default "F"*

Text=Field_text() *Campo testo di qualsiasi lunghezza*

List=Field_list() *Campo di tipo lista*

Tuple=Field_tuple() *Campo di tipo tuple*

Set=Field_set() *Campo di tipo set*

Dict=Field_dict() *Campo di tipo dict*

Data=Field_date() *Campo di tipo Date*

SetOf=Field_set_of({1,3,5,7}) *Accetta uno solo di questi valori 1,3,5,7*

Table la classe per la creazione delle tabelle di un database, questa classe assieme alle classi Field_type, costituiscono la stratificazione alla classe dbs che rende possibile la programmazione ad oggetti.

Modulo table.py

Questa è una classe astratta dove va definito solo il metodo __init__ , in cui andranno messi i vari campi, contiene inoltre i metodi di interfacciamento alla classe dbs che vengono wrappati dal nuovo oggetto Table, supporta la clausola with a puro scopo estetico.

Metodi Table:

dbs Ritorna il dbs in uso nella tabella

riassocia la maggior parte dei metodi di dbs
riassocia i magic method
__delitem__ __next__ __len__
Aggiunge i magic method:

__enter__ e __exit__ che permettono la clausola with
__str__ e __repr__ vanno ridefiniti a secondo le esigenze

Vediamo un esempio di dichiarazione di un oggetto derivato da Table.

```
from dbase.Table import Table
from dbase.Field_type import *
from dbase.Date import Date
class Dipendente(Table): #definisce una nuova classe ereditata da Table
    def __init__(self,table):
        super().__init__(table)  #chiama __init__   padre
        self.Matricola=Field_int()  # definisce Matricola di tipo Integer
        self.Cognome= Field_string(50) # Cognome stringa 50 caratteri
        self.Nome=Field_string(50)  # Nome stringa 50 caratteri
        self.Assunzione=Field_date() # data di Assunzione

     # Opzionale metodi __str__ e __repr_
    def __str__(self) : # aggiunge il metodo di formattazione di stringa
        return  "{} {} - Matricola: {} -  Assunzione: {}".format(self.Nome.value,
                self.Cognome.value,self.Matricola.value,self.Assunzione.value)

    __repr__=__str__  #__repr__ e __str__ cosi si equivalgono
```

Record la classe sotto agregatore questa classe astratta ha il solo scopo di raccogliere assieme i campi che hanno uno scopo comune.

Modulo Record.py

Record metodi

Come per Table gli oggetti derivati da Recoder andra ridefinito il metodo __init__ Record supporta la clausola with a puro scopo estetico.
I magic method __str__ e __repr__ se occorono dovranno essere ridefiniti.

Implementazione di un oggetto Record

```
from dbase.Table import Table
from dbase.Field_type import *
from dbase.Record import Record

class Recapito(Record):
    def __init__(self ):
        super().__init__()
        self.Telefono=Field_string(20)
        self.Cellulare=Field_string(20)
        self.Email=Field_string(25)
        self.Indirizzo=Field_string(50)

    def __str__(self):
        return  "Telefono: {} /n Cellulare: {} /n Email: {} /n Assunzione: {}".format(
Telefono.value, self.Cellulare.value, self.Email.value, self.Indirizzo.value)
```

Definire in una classe derivata da Table self.Recapito=Recapito() permette di accedere alle varie informazioni come un unico insieme.

objTab.Recapito.Telefono.value, objTab.Recapito.Cellulare.value

objTab.Recapito.Email.value, objTab.Recapito.Indirizzo.value

Table supporta un metodo di ricerca a chiave singola, intera o parziale, ma è una ricerca sequenziale.

Esempio:

if objTab.search('Paolino', objTab.Cognome)

Si posiziona sul record ritorna True se la ricerca ha esito, altrimenti False.
*usare **if objTab.next():** per continuare la ricerca, la chiave può essere anche parziale*
sono possibili 3 casi:
**no po* st*o*

if objTab.search('*lino', objTab.Cognome')

Come il vecchio DB III+, Dbase utilizza i file indice per una ricerca rapida, per mezzo della classe Idbs.

Modulo Idbs.py

La classe Idbs supporta la creazione di indici su uno o più campi, permette la ricerca su chiave intera o parziale.

Sintassi: Idbs(table, filename) per filename utilizzare estensione .index

Metodi:

index(*fields) Crea un indice in memoria su i campi indicati

reindex() Re indicizzazione

search(key, callback) Ricerca la chiave se la ricerca è positiva chiama la funzione di callback, per ogni occorenza della chiave.

next() Prossima ricerca

dbs Ritorna il dbs in uso (per eventuali espasioni)

table Ritorna l'oggetto Table (per eventuali espansioni)

flush() Salva il file indice.

use(name="") Carica in memoria un file indice salvato in precedenza

Magic method

__contains__ Ricerca veloce se un valore è presente nella chiave indicata ritorna True se presente e si posiziona sul Record esempio: "Rossi" in objIndex

__next__ Alternativa a objIndex.next() next(objIndex)

Esempio completo di riepilogo file testDbase.py

```
from dbase.Table import *
from dbase.Field_type import *
from dbase.Idbs import Idbs

class Agenda(Table):   # definisce la classe Agenda
    def __init__(self,table):
        super().__init__(table)
        self.Cognome=Field_string(30)
        self.Nome=Field_string(20)
        self.Telefono=Field_string(20)
        self.Email=Field_string(20)
    def __str__(self):
        return "{} {} - Telefono: {} - Email: {}".format(self.Nome.value,
              self.Cognome.value,self.Telefono.value,self.Email.value)

persona=Agenda('Agenda') # Crea un databasevuoto in memoria
persona.use() #carica il database  filename non indicato quindi il file sarà 'Agenda.tab'
ind=Idbs(persona,"Agenda-Cognome.index") #crea un oggetto indice per il  campo Cognome
ind2=Idbs(persona,"Agenda-Cognome-Nome.index") #secondo indice per Campi Cognome e Nome
ind.index(persona.Cognome) #indicizza il primo indice per Cognome
ind2.index(persona.Cognome, persona.Nome) #indice2 per Cognome e Nome
persona.seek_to_start()  #posizionati sul primo record
print('Chiave singola "Paolino"\n')
if "Paolino" in ind():  # Se il Cognome Paolino è presente
    print(persona)     # scrivi il record utilizza __str__
    while ind.next():  #se presente altra voce stampala
            print(persona)
else:   # ricerca senza esiti
    print("Chiave non presente")
print('Chiave singola parziale "De*"\n') #ricerca con chiave parziale
if "De*" in ind:
    print(persona)
    while ind.next(): # ricerca secondaria
            print(persona)
else:
    print("Chiave non presente") # ricerca senza esiti

print('Chiave Doppia  "Paolino-Paperino"\n')
if "Paolino-Paperino" in ind: #ricerca standart
    print(persona)
    while ind.next():
            print(persona)
else:
    print("Chiave non presente")
print('Chiave doppia parziale "Paolino*"\n')
if "Paolino*" in ind:
    print(persona)
    while ind.next():
```

```
        print(persona)
else:
   print("Chiave non presente")

def filtro(Noerr): #funzione di callback
   print(persona) if  Noerr else print("Chiave non presente")
   return

print('\nRicerca tramite callback chiave "De*"\n')
ind.search('De*',filtro) #Utilizzando search  con una chiave parziale e una funzione di callback
print('\nInserisco una chiave errata "Di*" ')
ind.search('Di*',filtro)
```

Un piccolo database di esempio file Agenda.tab


```
[{"3": "String", "0": "String", "2": "String", "1": "String"},
 {"1": "Betty", "0": "Boop", "3": "ss@099", "2": "309"},
 {"0": "Dasy", "1": "Duck", "2": "876", "3": "ss@987"},
{"1": "Pico", "0": "De Paperis", "3": "ss@11", "2": "666"},
{"1": "Paperon", "0": "De Paperoni", "3": "ss@04", "2": "567"},
 {"1": "Pippo", "0": "De Pippis", "3": "ss@02", "2": "234"},
{"1": "Duffy", "0": "Duck", "3": "ss@015", "2": "598"},
 {"1": "Mickey", "0": "Mouse", "3": "ss@05", "2": "555"},
{"0": "Paolino", "1": "Paperina", "2": "390", "3": "ss@028"},
{"0": "Paolino", "1": "Paperino", "2": "613", "3": "ss@012"},
{"1": "Gastone", "0": "Paperone", "3": "ss@19", "2": "288"},
{"1": "Jessica", "0": "Rabbit", "3": "ss@016", "2": "877"},
{"1": "Roger", "0": "Rabbit", "3": "ss@015", "2": "345"},
{"0": "Paolino", "1": "Tip", "2": "564", "3": "ss@876"},{"0": "Paolino", "1": "Tap", "2": "677",
"3": "ss@098"}]
```


Il primo record rappresenta le chiavi e i tipi di campi, le chiavi sono numeriche e l'ordine della dichiarazione dei campi rappresenta il modo in cui i campi si agganciano.

Output del file testDbase.py

```
Chiave singola "Paolino"

Paperina Paolino - Telefono: 390 - Email: ss@028
Paperino Paolino - Telefono: 613 - Email: ss@012
Tip Paolino - Telefono: 564 - Email: ss@876
Tap Paolino - Telefono: 677 - Email: ss@098

Chiave singola parziale "De*"

Pico De Paperis - Telefono: 666 - Email: ss@11
Paperon De Paperoni - Telefono: 567 - Email: ss@04
Pippo De Pippis - Telefono: 234 - Email: ss@02

Chiave Doppia  "Paolino-Paperino"

Paperino Paolino - Telefono: 613 - Email: ss@012

Chiave doppia parziale "Paolino*"

Paperina Paolino - Telefono: 390 - Email: ss@028
Paperino Paolino - Telefono: 613 - Email: ss@012
Tip Paolino - Telefono: 564 - Email: ss@876
Tap Paolino - Telefono: 677 - Email: ss@098


Ricerca tramite callback chiave doppia parziale "De*"

Pico De Paperis - Telefono: 666 - Email: ss@11
Paperon De Paperoni - Telefono: 567 - Email: ss@04
Pippo De Pippis - Telefono: 234 - Email: ss@02

Inserisco una chiave errata "Di*"
Chiave non presente
```

Il modulo dbRelaction mette in relazione due o più tabelle con un campo univoco.

Modulo dbRelaction.py

dbRelaction metodi

Sintassi: dbRelaction (self,*objdb:Idbs)

Lo scopo di questa classe è quella di posizionare una o più tabelle su un campo univoco comune, può essere utilizzata sd esempio, in una maschera video dove molti campi appartengono a tabelle diverse ma che abbiano un campo univoco in comune esempio il numero di matricola di un dipendente o per sicronizzare più tabelle.

Quindi dbRelaction sincronizza i vari oggetti indice sui record a cui la chiave corrisponde.

Magic method:

__contains__ ***matricola in dbr*** posiziona tutti i file indice sullo stesso numero di matricola.

Metodi:

append_blank() appende un record vuoto su tutte le tabelle interessate.
flush() salva tutti i vari dbs e gli indici collegati.
reindex() reindicizza tutti gli oggetti indice.

modulo dbMath.py

Il modulo dbMath supporta le più comuni operazioni matematiche su un database
Sintassi: ***dbm= dbMath(objTable)***.

Total(field,Filter=None) effettua la somma di tutti i valori del campo field, Filter eventuale funzione di callback di controllo.

Average(field,Filter=None) Effettua la media di tutti i valori del campo field, Filter eventuale funzione di callback di controllo.

Min(field) Calcola il valore minimo del campo field

Max(field) Calcola il valore massimo del campio field

Math_esempio.py

```
from dbase.Table import *
from dbase.Field_type import *
from dbase.dbMath import dbMath
import random

class Math_Es(Table):
    def __init__(self,table):
        super().__init__(table)
        self.Integer=Field_int()
        self.Float=Field_float()
        self.Complex=Field_complex()
        self.Rinteger=Field_int(Range=(3,1200))
    def __str__(self):
        return "Totale Integer={} - Totale Float={} - Totale Complex={} – Totale
                    Integer(3,1200)={}:".format(self.Integer.value, self.Float.value, self.Complex.value,
                    self.Rinteger.value)

db=Math_Es('Math_es') #nuovo database vuoto

for i in range(10): # riempio il database con valori a caso per  10 elementi
    db.append_blank()  #nuovorecord vuoto
    db.Integer.value= random.randrange(7,77)
    db.Float.value= random.randrange(45,176.0)*0.1567
    db.Complex.value= random.randrange(8,100.0)*(6+62j)
    db.Rinteger.value= random.randrange(3,1200)

m=dbMath(db) # modulo matematico ancorato a  “Math_es”
print('Totale Integer={}'.format(m.total(db.Integer)))
print('Totale Float={}'.format(m.total(db.Float)))
print('Totale Complex={}'.format(m.total(db.Complex)))
```

```
print('Totale Integer(3,1200)={}'.format(m.total(db.Rinteger)))
print('_'*50)
print('Max Integer={}'.format(m.max(db.Integer)))
print('Max Float={}'.format(m.max(db.Float)))
print('Max Complex={}'.format(m.max(db.Complex)))
print('Max Integer(3,1200)={}'.format(m.max(db.Rinteger)))
print('_'*50)
print('Min Integer={}'.format(m.min(db.Integer)))
print('Min Float={}'.format(m.min(db.Float)))
print('Min Complex={}'.format(m.min(db.Complex)))
print('Min Integer(3,1200)={}'.format(m.min(db.Rinteger)))
print('_'*50)
print('Average Integer={}'.format(m.average(db.Integer)))
print('Average Float={}'.format(m.average(db.Float)))
print('Average Complex={}'.format(m.average(db.Complex)))
print('Average Integer(3,1200)={}'.format(m.average(db.Rinteger)))
```

Output: Math_esempio.py

```
Totale Integer=406
Totale Float=151.0588
Totale Complex=(3048+31496j)
Totale Integer(3,1200)=6587

_____________________________________________________
Max Integer=70
Max Float=23.1916
Max Complex=(582+6014j)
Max Integer(3,1200)=1062

_____________________________________________________
Min Integer=8
Min Float=7.835
Min Complex=(60+620j)
Min Integer(3,1200)=204

_____________________________________________________
Average Integer=40.6
Average Float=15.105879999999999
Average Complex=(304.8+3149.6j)
Average Integer(3,1200)=658.7
```

Tranelli in cui non cadere

>>> a=persona (persona =Agenda("Agenda"))

>>> a

Tap Paolino Telefono: 677 Email: ss@098 *a è un alias di persona non una stringa*

>>> type(a)

<class '__main__.Persona'>

>>> b=str(a**)** o str(persona) *corretto se si vuole una stringa, __str__ deve essere definito*

>>> a=persona.Cognome *a è un alias di persona.Cognome non il suo valore, a.value Corretto, print(a) Corretto sfrutta il metodo __str__ del campo altrimenti print(a.value)*

>>> a= persona.Cognome.value *Corretto a contiene il contenuto del campo*

persona .Cognome.value='De Paperoni' *Corretto il campo è modificato*

persona.Cognome='De Paperoni' *Non corretto ora l'etichetta persona.Cognome è una stringa*

Esempio : riordina il database e stampa il contenuto

```
persona.sort(persona.Cognome) # ordina il database per Cognome
# metodo 1  stampa l'intero database  (stile file)
persona.seek_to_start()   # vai al primo record
while not persona.eof():  # cicla fino  alla fine del database
   with persona as r: # ha solo la funzione di rendere il sorgente più chiaro
      print("Nome=",r.Nome) # stampa singolarmente i campi
      print("Cognome=",r.Cognome)
      print("Telefono=",r.Telefono)
      print("Email=",r.Email)
      print("-"*20)
      print(r) # stampa con il formato ufficiale del record  __str__ deve esere definito
      print("-"*20)
     next(persona)  # vai al prossimo record


#metodo 2 stampa l'intero database (stile più pythonico)
for r in persona:
      print("Nome=",r.Nome) # stampa singolarmente i campi
      print("Cognome=",r.Cognome)
      print("Telefono=",r.Telefono)
      print("Email=",r.Email)
      print("-"*20)
      print(r) # stampa con il formato ufficiale del record __str__ deve esere definito
      print("-"*20)
```

Per accedere fisicamente al valore di un campo in lettura e in scrittura, utilizzare sempre la proprietà value.

```
from dbase.Table import Table
from dbase.Field_type import *
from dbase.Date import Date

class test(Table): # definisce i campi di un database
    def __init__(self,table):
      super().__init__(table)
      self.Nome=Field_string(30)
      self.Carattere=Field_char(valid_char=('m,f','f'))
      self.Eta=Field_int(Range=(0,150))
      self.Complesso=Field_complex(Range=(2+3j,7+0j))
      self.Float=Field_float(Range=(4.0,7.0))
      self.Intero=Field_int(Range=(5,25))
      self.Boolean=Field_bool()
      self.List=Field_list()
      self.Tuple=Field_tuple()
      self.Set=Field_set()
      self.Dict=Field_dict()
      self.Date=Field_date()
d=Date.today() #data odierna
User=test('test')  #crea il database
User.append_blank()    #appende un record vuoto
User.Nome.value='Topolino'   # scrive i campi
User.Carattere.value="mio" # "m" è un carattere valido viene controllato solo il primo carattere
User.Eta.value=50
User.Complesso.value=(2+3j)
User.Float.value=(6.5)
User.Intero.value=5
User.Boolean.value=True
User.List.value=[4,5,6]
User.Tuple.value=(4,8)
User.Set.value={1,2,3}
User.Dict.value={1:3,5:6}
User.Date.value=d
print("Nome =",User.Nome)  #stampa i campi del record utilizza __str__  alternativa User.Nome.value
print("Catattere =",User.Carattere)
print("Eta =",User.Eta)
print("Complesso =",User.Complesso)
print("Float =",User.Float)
print("Intero =",User.Intero)
print("Boolean =",User.Boolean)
print("List =",User.List)
print("Tuple =",User.Tuple)
print("Set =",User.Set)
print("Dict =",User.Dict)
print("Date =",User.Date)
User.flush()  # scrive il database test.tab
```

Output:

Nome = Topolino
Carattere = m
Eta = 50
Complesso = (5+3j)
Float = 6.5
Intero = 5
Boolean = True
List = [4, 5, 6]
Tuple = (4, 8)
Set = {1, 2, 3}
Dict = {1: 3, 5: 6}
Date = 2022-04-06

dbMath esempio

```
from dbase.Table import *
from dbase.Field_type import *
from dbase.dbMath import dbMath
import random

class Math_Es(Table): #definisce la struttura del database
    def __init__(self,table):
        super().__init__(table)
        self.Integer=Field_int()
        self.Float=Field_float()
        self.Complex=Field_complex()
        self.Rinteger=Field_int(Range=(3,1200,0))
    def __str__(self):
        return " Integer={} - Float={} - Complex={} - Integer(3,350,0)={}:".format(self.Integer. value,
                self.Float.value, self.Complex.value, self.Rinteger.value)

 db=Math_Es('Math_es')
for i in range(10): # Inserisco nel database 10 elementi con valori a caso
    db.append_blank()
    db.Integer.value= random.randrange(7,77)
    db.Float.value= random.randrange(45,176.0)*0.1567
    db.Complex.value= random.randrange(8,100.0)*(6+62j)
   db.Rinteger.value= random.randrange(3,1200)
m=dbMath(db) # Lo collego al modulo matematico
print('Totale Integer={}'.format(m.total(db.Integer))) #stampo i valori
print('Totale Float={}'.format(m.total(db.Float)))
print('Totale Complex={}'.format(m.total(db.Complex)))
print('Totale Integer(3,1200,0)={}'.format(m.total(db.Rinteger)))
print('_'*50)
print('Max Integer={}'.format(m.max(db.Integer)))
print('Max Float={}'.format(m.max(db.Float)))
print('Max Complex={}'.format(m.max(db.Complex)))
print('MaxInteger(3,1200,0)={}'.format(m.max(db.Rinteger)))
print('_'*50)
print('Min Integer={}'.format(m.min(db.Integer)))
```

```
print('Min Float={}'.format(m.min(db.Float)))
print('Min Complex={}'.format(m.min(db.Complex)))
print('Min Integer(3,1200)={}'.format(m.min(db.Rinteger)))
print('_'*50)
print('Average Integer={}'.format(m.average(db.Integer)))
print('Average Float={}'.format(m.average(db.Float)))
print('Average Complex={}'.format(m.average(db.Complex)))
print('Average Integer(3,1200)={}'.format(m.average(db.Rinteger)))
```

Output:

Totale Integer=503
Totale Float=153.56599999999997
Totale Complex=(3702+38254j)
Totale Integer(3,1200)=4290

Max Integer=75
Max Float=25.5421
Max Complex=(558+5766j)
Max Integer(3,1200)=1043

Min Integer=13
Min Float=7.835
Min Complex=(66+682j)
Min Integer(3,1200)=119

Average Integer=50.3
Average Float=15.356599999999997
Average Complex=(370.2+3825.4j)
Average Integer(3,1200)=429.0

Query in Dbase

Negli esempi predenti abbiamo gia visto delle ricerche semplici. Nelle query si vede la versabilità di Dbase:

Si voglia dare un aumento di stipendio di € 50.00 a tutti i dipendenti di 1 Livello con 3 o più figli, i campi della tabella dipedenti che ci interessano sono stipendio float, livello int, figli int.

Metodo 1:

```
for dip in dipendenti:
    if dip.livello.value==1 and dip.figli.value>=3: dip.stipendio.value+=50.00
```

Metodo 2:

```
if (dipendenti.search(1, dip.livello) and dipendenti.figli.value>=3):
   dipendenti.stipendio.value +=50.00
   while (dipendenti.next() and dipendenti.figli.value>=3): dipendenti.stipendio.value +=50.00
```

Con indice:

Metodo 1:

```
liv=Idbs(dipendenti, "Livello.index") #Crea l'oggetto indice
liv.index(dipendenti.livello) # Indicizza su livello

if  (1 in liv  and dipendenti.figli.value>=3): # start ricerca
    dipendenti.stipendio.value +=50.00
    while (liv.next()  and dipendenti.figli.value >=3): #continua  la ricerca
        dipendenti.stipendio.value +=50.00
```

Metodo 2:

```
liv=Idbs(dipendenti.livello, "Livello.index") #Crea il file indice

def aumento(): # callback
    if dipendenti.figli.value >=3: dipendenti.stipendio.value +=50.00

liv.search(1,aumento) #per ogni dipendente di livello 1 con 3 o più figli applica l'aumento di stipendio
```

Espandere Dbase

Dbase è facilmene espandibile, un piccolo esempio, la funzione append_from appende dei campi di una tabella in un'altra contenente gli stessi campi, le due liste di campi devono avere i campi nello stesso ordine.

modulo append_from.py

```
def append_from(sfields:list, dfields:list, Filter =None):
    """ sfields lista di campi tabella sorgente es. [persona.Cognome, persona.Nome]
        dfields lista campi tabella di destinazione es. [dest.Cognome, dest.Nome]
        callback eventuale funzione filtro o di convalida"""

    db1=sfields[0].dbs # dbs di riferimento
    db2=dfields[0].dbs
    rpos=db1.tell()  #salva la posizione del puntatore
    db1.seek_to_start() # primo record del sorgente

    def sub_Call(): #sotto funzione
        nonlocal db1,db2,sfields,dfields
        db2.append_blank()
        for i,f in enumerate(sfields):
            dfields[i].value=f.value

    while not db1.eof(): # cicla su tutto il dbs
        if Filter is not None: #se callback è presente
            if Filter (sfields):  #ritorna True se campi è validi
                sub_Call()
        else:   #senza callback
            sub_Call()
        next(db1)  #Prossimo record

    db1.seek(pos)  # ripristina il puntatore
```

Questa versione di append_from si limita a 2 tabelle una sorgente e l'altra di destinazione, ma si può considerare di creare una versione che prenda i campi sorgente da più tabelle per immetterli in unica destinazione.

Questo tutorial si conclude qui, Dbase ha molte potenzialità di utilizzo, dai programmi di gestione, esempio spese famigliari, ai giochi, programmi di quiz a scopo didattico o per puro divertimento con gli amici, ecc.. .

I vantaggi di Dbase sono molti, è semplice, veloce (lavora completamente in memoria), un interfaccia ad oggetti che ne facilità la gestione.