Python Sqlite3-tietokantakirjasto

Luodaan ensin yhteysolio, joka edustaa tietokantaa. Tässä esimerkissä tietokantojen taulut tallennetaan tiedostoon:

```
conn = sqlite3.connect('/tmp/example.sqlite3')
```

Kun olet luonut yhteysolion, voit luoda sille kursoriolion ja kutsua sen **execute**-metodia suorittaaksesi SQL-komentoja:

```
c = conn.cursor()
# Luo taulu
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
# Lisataan uusi rivi tauluun
c.execute("""insert into stocks
      values ('2006-01-05','BUY','RHAT',100,35.14)""")
# Tallenna tietokanta ja sulje kursori
conn.commit()
c.close()
```

Muuttujat välitetään Pythonista SQL-operaatioille parametrien avulla, yleensä vektorimuodossa **execute**-metodille. Kysymysmerkki toimii tietokantakyselyn paikkamerkkinä.

Esimerkki.

```
symbol = 'IBM'
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)
```

Laajempi esimerkki.

```
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
  ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
  ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
):
c.execute('insert into stocks values (?,?,?,?,?)', t)
```

Tietojen lukemiseksi SELECT-lausekkeelta voit joko käydä lävitse kursoria iteraattorina, kutsua kursorin **fetchone**-metodia lukeaksesi yhden rivin kerrallaan, tai kutsua **fetchall**-metodia lukeaksesi listan osumista.

Tämä esimerkki käyttää iteraattoria:

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...    print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
>>>
```

Kirjaston funktiot ja attribuutit

Funktio **sqlite3.connect(database[,timeout, isolation_level, detect_types, factory])**

Luo yhteyden Sqlite-tietokantaan. Voit käyttää nimeä **:memory:** avataksesi tietokantayhteyden tietokantaan, joka sijaitsee kiintolevyn sijasta keskusmuistissa. Prosessin käsitellessä Sqlite-tietokantaa, tietokanta lukitaan kunnes toimenpide on suoritettu. Parametri **timeout** määrittää kuinka pitkään yhteyden on odotettava lukituksen vapautumista, kunnes se nostaa poikkeuksen. Oletusarvo on viisi sekuntia. **Connect**-funktion toinen valinnainen parametri on eristystasoparametri **isolation_level**. Tästä lisää myöhemmin. Oletusarvoisesti Sqlite tukee ainoastaan seuraavia tietotyyppejä:

TEXT	INTEGER	FLOAT	BLOB	NULL

Jos haluat käyttää muita tietotyyppejä, on sinun lisättävä tuki niille itse. Tietotyyppiparametrin määritys ja erikoistettujen muuntimien rekisteröinti kirjaston **register_converter**-funktiolla antaa sinulle mahdollisuuden tähän. **Connect**-funktion neljäs valinnainen parametri on nimeltään **detect_types**. Oletusarvo tälle parametrille on 0, ei tyyppitunnistusta. Voit asettaa sille arvoksi minkä tahansa vakioiden **PARSE_DECLTYPES** ja **PARSE_COLNAMES** yhdistelmän kytkeäksesi tunnistuksen päälle.

Vakio **sqlite3.PARSE_DECLTYPES**

Sarakkeesta pyritään löytämään sarakkeen määrittelynimi. Connect-funktio etsii ensimmäisen sanan määritellystä tyypistä, eli esimerkiksi kokonaislukutyypille ensisijaisena avaimena integer primary key, se löytää sanan "integer", ja numerotyypille "number 10" se löytää sanan "number". Sen jälkeen se etsii sarakkeelle muunnossanakirjasta arvon ja käyttää tälle tyypille rekisteröityä muunnosfunktiota.

Vakio **sqlite3.PARSE_COLNAMES**

Tämän vakion antaminen **connect**-funktion neljäntenä parametrina saa Sqlite-kirjaston etsimään sarakkeen nimen kustakin sarakkeesta, jonka se palauttaa. Se etsii siitä merkkijonoa, joka on muotoiltu tyyliin [mytype], ja päättää sen perusteella että "mytype" on sarakkeen tyyppi. Se pyrkii löytämään mytypeä vastaavan alkion muunnossanastosta ja käyttää sen jälkeen sieltä löytämäänsä muunnosfunktiota palauttaakseen arvon. Sarakkeen nimi, joka löytyy kursorimäärittelystä **cursor.description** on yksinkertaisuudessaan ensimmäinen sana sarakkeen nimestä. Jos käytät vaikkapa määrittelyä tyyliin **'as "x [datetime]"'** SQL-komennossa, niin tuloksena on sarakkeen nimestä kaikki ne merkit, jotka edeltävät ensimmäistä välilyöntiä: tässä tapauksessa sarakkeen nimeksi tulee yksinkertaisesti "x". Oletusarvoisesti sqlite3-kirjasto käyttää **Connection**-luokkaa tietokantaan yhdistettäessä. Voit kuitenkin halutessasi periyttää Connection-luokan ja kytkeä funktion käyttämään tekemääsi luokkaa. Katso lisätietoa kappaleesta "Pythonin ja Sqliten tyypit". Jos haluat itse määrittää välimuistiin kerättävien komentojen lukumäärän yhteyttä kohden, voit asettaa **cached_statements**-parametrin arvon. Oletusarvoisesti välimuistiin kerättävien komentojen määrä on 100.

Muuntimen rekisteröinti

Funktio **sqlite3.register_converter(typename, callable)**

Tämä metodi rekisteröi kutsuttavan funktion tietokannasta haetun tavumuotoisen tiedon muuttamiseksi Pythonin muuttujatyypiksi. Kyseistä funktiota kutsutaan kaikkien niiden tietokannasta saatujen arvojen kohdalla, jotka ovat funktion ensimmäisenä parametrina annettua tyyppiä. Katso lisätietoa kohdasta "Parametrin tyyppitunnistus connect-funktiolle". Huomaa, että tyyppinimien kirjoitusasun tulee vastata toisiaan.

Adapterin rekisteröinti

Funktio **sqlite3.register_adapter(type, callable)**

Tämä metodi rekisteröi adapterin Pythonin muuttujatyypin muuttamiseksi tietokannan tukemaksi tyypiksi. Muuttujatyyppi annetaan metodin ensimmäisenä parametrina. Metodille toisena parametrina annettava kutsuttava funktio on yksiparametrinen, ja sen tulee palauttaa jokin seuraavien tyyppien muuttuja-arvoista:

int	**long**	**float**	**str** (utf8-koodattuna)	**unicode**	**buffer**

Funktio **sqlite3. complete_statement(sql)**

Palauttaa arvon **True**, mikäli parametrin merkkijono sisältää yhden tai useamman muodollisesti kelvollisen puolipisteeseen päättyvän SQL-lausekkeen. Tätä funktiota voit käyttää esimerkiksi minimaalisen SQL-komentotulkin tekemiseen:

```
import sqlite3
con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()
buffer = ""
print "Enter your SQL commands to execute in SQLite."
print "Enter a blank line to exit."
while True:
  line = raw_input()
  if line == "":
    break
  buffer += line
  if sqlite3.complete_statement(buffer):
    try:
      buffer = buffer.strip()
      cur.execute(buffer)
      if buffer.lstrip().upper().startswith("SELECT"):
        print cur.fetchall()
    except sqlite3.Error, e:
      print "An error occurred:", e.args[0]
    buffer = ""
con.close()
```

Funktio **sqlite3.enable_callback_tracebacks(flag)**

Oletusarvoisesti sqlite3-kirjasto ei tuota virheilmoitusten pinolistauksia. Funktion arvo **True** kytkee pinolistauksen käyttöön takaisinkutsufunktioilta standardivirtaan sys.stderr. Vastaavasti funktion arvo **False** poistaa ominaisuuden käytöstä.

Yhteysoliot

Yhteysolioilla on seuraavat attribuutit ja metodit:

Attribuutti **Connection. isolation_level**

> Asettaa tai palauttaa käytössä olevan eristystason. Käytä arvoa None automaattiseen tallentamiseen tai yhtä seuraavista: **DEFERRED**, **IMMEDIATE** tai **EXCLUSIVE**. Katso lisätietoa kappaleesta Tiedonsiirron hallinta.

Funktio **Connection.cursor([cursorClass])**

> Tämä metodi luo kursorin. Valinnaisena parametrina on mahdollista antaa **sqlite3.Cursor**-luokasta periytetty kursoriluokka.

Funktio **Connection.commit()**

> Tämä metodi tallentaa tiedoston ja päivittää tietokannan tilan.

Funktio **Connection.rollback()**

> Tämä metodi peruu muutokset, jotka tehtiin viimeisen **commit**-kutsun jälkeen.

Funktio **Connection.close()**

> Tämä metodi sulkee yhteyden tietokantaan. Metodi ei kutsu **commit**-metodia.

Funktiot **Connection.execute**, **Connection.executemany** ja **Connection.executescript**

> Nämä epästandardit metodit luovat välittömän kursoriolion kutsumalla **cursor**-metodia, jonka jälkeen kukin kutsuu vastaavaa kursorin **executeXX**-metodia annetuilla parametreilla.

Oman SQL-funktion luominen

Funktio **Connection.create_function(name, num_params, func)**

> Tämä metodi luo käyttäjän määrittelemän funktion, jota voit myöhemmin käyttää annetun SQL-funktion nimen avulla. Parametrit ovat SQL-funktion nimi, funktiolle välitettävien parametrien lukumäärä sekä kutsuttava funktio. Kutsuttaessa ensinmainittua SQL-funktiota, funktiokutsu parametreineen välitetään viimeksimainitulle Python-funktiolle. Funktio voi palauttaa minkä tahansa seuraavista Sqliten tukemista tietotyypeistä:
>
> | unicode | str | int | long | float | buffer | None |
> |---|---|---|---|---|---|---|
>
> Esimerkki.

```
import sqlite3
import md5
def md5sum(t):
  return md5.md5(t).hexdigest()
con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", ("foo",))
print cur.fetchone()[0]
```

Oman aggregaattifunktion luominen

Funktio **Connection.create_aggregate(name, num_params, aggregate_class)**

Tämä metodi luo käyttäjän määrittelemän aggregaattifunktion. Ensimmäinen parametri on aggregaatin nimi. Toinen parametri on tälle funktiolle annettavien parametrien lukumäärä. Kolmas parametri on aggregaattiluokka.

Aggregaattiluokan tulee toteuttaa metodi **step**, jonka parametrien määrä edellä määritettiin, ja metodi **finalize**, joka palauttaa aggregaatin tuottaman lopputuloksen. **Finalize**-metodi voi palauttaa minkä tahansa seuraavista Sqliten tukemista tietotyypeistä:

unicode	str	int	long	float	buffer	None

Esimerkki.

```
import sqlite3
class MySum:
  def __init__(self):
    self.count = 0
  def step(self, value):
    self.count += value
  def finalize(self):
    return self.count
con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print cur.fetchone()[0]
```

Oman lajittelualgoritmin luominen

Funktio **Connection.create_collation(name, callable)**

Luo lajittelualgoritmin. Parametrit ovat algoritmin nimi ja siihen yhdistettävä funktiokutsu. Kutsuttavalle funktiolle välitetään kaksi merkkijonoargumenttia. Sen tulee palauttaa arvo -1, 0 tai +1 seuraavasti:

-1	jos ensimmäinen on järjestyksessä toisen jälkeen,
0	jos ensimmäinen on yhtäsuuri kuin toinen ja
+1	jos ensimmäinen on järjestyksessä ennen toista.

Huomaa, että tämä metodi hallinnoi myös SQL-kutsujen lajittelujärjestystä. Kutsuttava funktio saa parametrinsa Pythonin tavuvirtana, joka tavallisesti koodataan utf8-koodauksella. Seuraava esimerkki näyttää kuinka oman lajittelualgoritmin saa lajittelemaan niin sanotusti "väärin päin":

```
import sqlite3
def collate_reverse(string1, string2):
  return -cmp(string1, string2)
con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)
cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
  print row
con.close()
```

Poista lajittelualgoritmi kutsumalla metodia **create_collation** parametrilla **None**.

```
con.create_collation("reverse", None)
```

Funktio **Connection.interrupt()**

Voit kutsua tätä metodia toisesta säikeestä pysäyttääksesi kaikki haut, jotka mahdollisesti ovat yhteydellä käynnissä. Haut perutaan ja kutsuva funktio saa poikkeuksen.

Funktio **Connection.set_authorizer(authorizer_callback)**

Tämä metodi määrittää takaisinkutsufunktion käytettäväksi tietokannan muutosten autorisointiin.

Funktio **Connection.set_progress_handler(handler, n)**

Tämä metodi määrittää väliaikatietoa tuottavan funktion. Takaisinkutsufunktiota kutsutaan aina muuttujan määräämällä vakiovälillä suoritettaessa Sqlite-virtuaalikoneen toimintoja. Funktiota voi käyttää pitkään kestävälle operaatiolle esimerkiksi käyttöliittymän päivittämiseen.
Poista käsittelijä parametrin arvolla **None**.

Lisäosien lataaminen

Funktio **Connection.enable_load_extension(enabled)**

Tämä metodi sallii tai estää Sqlite-tietokantamoottoria lataamasta Sqliten lisäosia jaetuista kirjastoista. Sqlite-lisäosat voivat määritellä uusia funktioita, aggregaatteja ja kokonaisten virtuaalitaulujen toteutuksia. Yksi hyvä esimerkki lisäosasta on laajennettu kokotekstihaku. Tämä lisäosa kuuluu vakiona sqlite3-asennukseen.

```
import sqlite3con = sqlite3.connect(":memory:")
# Sallii lisaosien lataamisen
con.enable_load_extension(True)
# Lataa lisaosan laajennetulle kokotekstihaulle
con.execute("select load_extension('./fts3.so')")
# Vaihtoehtoisesti voit ladata lisaosan kayttamalla API-kutsua:
# con.load_extension("./fts3.so")
# Poistetaan jalleen lisaosien latausmahdollisuus
con.enable_load_extension(False)
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
  insert into recipe (name, ingredients)
    values ('broccoli stew', 'broccoli peppers cheese tomatoes');
  insert into recipe (name, ingredients)
    values ('pumpkin stew', 'pumpkin onions garlic celery');
  insert into recipe (name, ingredients)
    values ('broccoli pie', 'broccoli cheese onions flour');
  insert into recipe (name, ingredients)
    values ('pumpkin pie', 'pumpkin sugar flour butter');
  """)
for row in con.execute(
  "select rowid, name, ingredients from recipe where name match 'pie'"):
  print row
```

Funktio **Connection.load_extension(path)**

Tämä metodi lataa Sqlite-lisäosan jaetuista kirjastoista. Sinun on sallittava lisäosien lataaminen ennen kuin voit käyttää tätä rutiinia.

Rivin luontiolio

Attribuutti **Connection.row_factory**

Tämä on rivin luontiolio tulosten tuottamiseen.
Attribuutti asetetaan osoittamaan funktioon. Funktion parametreina ovat kursori sekä alkuperäinen rivi vektorina. Funktion tulee palauttaa todellinen tulosrivi. Voit tällä tavoin palauttaa esimerkiksi olion, joka käsittelee sarakkeita niiden nimillä.
Esimerkki.

```
import sqlite3
def dict_factory(cursor, row):
  d = {}
  for idx, col in enumerate(cursor.description):
    d[col[0]] = row[idx]
  return d
con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print cur.fetchone()["a"]
```

Jos vektorin palauttaminen ei riitä, ja jos haluat käsitellä sarakkeita niiden nimen perusteella, on paras valita rivin luontiolioksi optimoitu **sqlite3.row**-tyyppi. **Row**-tyyppi mahdollistaa niin järjestysnumeroon perustuvan samoin kuin ei-merkkikokoherkän nimeen perustuvan sarakkeiden käsittelyn. Tämä on todennäköisesti parempi tapa kuin oman sanaston käyttö tai tietokannan rivin **db_row**-metodiin perustuva ratkaisu.

Attribuutti **Connection.text_factory**

Käyttämällä tätä attribuuttia voit hallita, mitkä oliot palautetaan tietotyypille TEXT. Oletuksena tämä attribuutti on asetettu arvoon **unicode** ja sqlite3-kirjasto palauttaa TEXT-tietotyypille unicode-olion. Jos tämän sijasta haluat sen palauttavan tavumuotoisen merkkijonon, voit asettaa sen arvoon **str**. Kolmas vaihtoehto on antaa attribuutille arvo **sqlite3.OptimizedUnicode**. Voit myös asettaa tämän attribuutin arvoksi jonkin muun kutsuttavan funktion. Tällöin funktion parametrina tulee olla tavumuotoinen merkkijono ja sen tulee palauttaa tulosolio.
Esimerkki sivuutetaan.

Attribuutti **Connection.total_changes**

Palauttaa muutettujen, lisättyjen ja poistettujen tietokannan rivien yhteismäärän alkaen hetkestä, jolloin tietokantayhteys luotiin.

Kursorioliot

Kursoriolion ilmentymillä on seuraavat attribuutit ja metodit:

Funktio **Cursor.execute(sql[,parameters])**

Suorittaa SQL-komennon. SQL-komento voi olla parametrisoitu tarkoittaen, että se sisältää paikkamerkkejä pelkkien SQL-komentojen sijasta. Sqlite3-kirjasto tukee kahdenlaisia paikkamerkkejä: kysymysmerkki ja nimetty paikkamerkki. Tämä esimerkki näyttää, miten kysymysmerkkiä käytetään parametreissa:

```
import sqlite3
con = sqlite3.connect("mydb")
cur = con.cursor()
who = "Yeltsin"
age = 76
cur.execute(
  "select name_last, age from people where name_last=? and age=?", (who,
age))
print cur.fetchone()
```

Tämä esimerkki käyttää nimettyjä paikkamerkkejä:

```
import sqlite3
con = sqlite3.connect("mydb")
cur = con.cursor()
who = "Yeltsin"
age = 76
cur.execute(
  "select name_last, age from people where name_last=:who and age=:age",
  {"who": who, "age": age})
print cur.fetchone()
```

Execute-funktio suorittaa yksittäisen SQL-komennon. Käytä **executescript**-funktiota, jos haluat suorittaa useita SQL-komentoja yhdellä kutsulla.

Funktio **Cursor.executemany(sql, seq_of_parameters)**

Suorittaa SQL-komennon kaikkia parametrin parametrijonoja tai kuvauksia vasten. Sqlite3-kirjasto mahdollistaa myös iteraattorin käyttämisen parametrien luontiin vektorin käyttämisen sijasta.

Esimerkki.

```
import sqlite3
class IterChars:
  def __init__(self):
    self.count = ord('a')
  def __iter__(self):
    return self
  def next(self):
    if self.count > ord('z'):
      raise StopIteration
    self.count += 1
    return (chr(self.count - 1),) # tama on vektori
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
cur.execute("select c from characters")
print cur.fetchall()
```

Tässä on lyhyempi esimerkki generaattoria käyttäen:

```
import sqlite3
def char_generator():
  import string
  for c in string.letters[:26]:
    yield (c,)
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
cur.executemany("insert into characters(c) values (?)", char_generator())
cur.execute("select c from characters")
print cur.fetchall()
```

Funktio **Cursor.executescript(sql_script)**

Tämä on epästandardi metodi useiden SQL-lauseiden suorittamiseen kerrallaan. Se suorittaa **commit**-lauseen ensin, jonka jälkeen se suorittaa parametrinaan saamansa SQL-komentosarjan. Esimerkki.

```
import sqlite3
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
  create table person(
     firstname,
     lastname,
     age
  );
  create table book(
     title,
     author,
     published
  );
  insert into book(title, author, published)
  values (
     'Dirk Gently''s Holistic Detective Agency',
     'Douglas Adams',
     1987
  );
  """)
```

Funktio **Cursor.fetchone()**

> Tuottaa seuraavan rivin kyselyn tulosjoukosta, palauttaen yksinkertaisen jonon tai arvon **None** mikäli tulosjoukko on loppuunkäsitelty.

Funktio **Cursor.fetchmany([size=cursor.arraysize])**

> Tuottaa seuraavan joukon tulosrivejä kyselyn tuloksesta, palauttaen luettelon. Tyhjä luettelo palautetaan, mikäli enempää rivejä ei ole saatavilla.
>
> Yhdellä kutsulla tuotettavien rivien yhteismäärä on määritelty nimetyllä parametrilla **size**. Jos sitä ei ole annettu, kursorin taulukonkoko määrittelee tuotettavien rivien lukumäärän.

Funktio **Cursor.fetchall()**

> Tuottaa kaikki jäljelläolevat rivit kyselyn tulosjoukosta, palauttaen luettelon. Huomaa, että kursorin taulukonkoko-attribuutti saattaa vaikuttaa operaation toimivuuteen. Tyhjä luettelo palautetaan, mikäli lisää rivejä ei ole saatavilla.

Funktio **Cursor.rowcount**

> Muutosten laskuri. DELETE-lauseille Sqlite tuottaa **rowcount**-muuttujaksi 0, mikäli suoritat komennon **delete from table** ilman valintaehtoja. **Executemany**-lauseille muutosten yhteenlaskettu määrä ilmoitetaan muuttujassa **rowcount**. Python DB Apin määritysten mukaisesti rivilaskuriattribuutti rowcount on -1 kun yhtään **executeXX**-komentoa ei tulla suorittaneeksi kursorissa tai kun rivilaskuri **rowcount** edellisestä operaatiosta ei ole liittymän pääteltävissä. Tämä pitää sisällään SELECT-lauseet, sillä kyselyn tuottamien rivien lukumäärää ei voi päätellä ennen kuin kaikki rivit on tuotettu.

Attribuutti **Cursor.lastrowid**

> Tämä vain luettava attribuutti tarjoaa viimeiseksi muutetun rivin identiteettinumeron. Se asetetaan vain silloin kun suoritat INSERT-lausekkeen käyttäen **execute**-metodia. Muille kuin INSERT-operaatiolle, tai kun **executemany**-metodia kutsutaan, viimeisen rivin identiteettinumero asetetaan arvoon **None**.

Pythonin ja Sqliten tyyppien käsittelystä

Sqlite3 tukee oletuksena seuraavia tietotyyppejä:

NULL	INTEGER	REAL	TEXT	BLOB

Seuraavat Pythonin tyypit voi siten lähettää Sqlitelle ilman ongelmia:

Python-tyyppi	Sqlite-tyyppi
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (utf8-koodattuna)	TEXT
unicode	TEXT
buffer	BLOB

Seuraavassa on esitettynä miten sqlite3-tyypit oletusarvoisesti muutetaan Pythonin tyypeiksi:

Sqlite-tyyppi	Python-tyyppi
NULL	None
INTEGER	int tai long (koosta riippuen)
REAL	float
TEXT	riippuen tekstin luontioliosta (unicode oletusarvoisesti)
BLOB	buffer

Sqlite3-kirjaston tyyppijärjestelmä on laajennettavissa kahdella tapaa: Toisaalta voit tallentaa muita Python-tyyppejä sqlite3-tietokantaan käyttämällä olioiden adaptointia, ja toisaalta voit antaa sqlite3-kirjaston muuntaa sqlite3-tyyppejä Python-tyypeiksi käyttäen muuntimia.

Adapterien käyttö muiden Python-tyyppien tallentamiseksi sqlite3-tietokantaan

Kuten aikaisemmin kuvattiin, sqlite3 tukee oletusarvoisesti ainoastaan rajoitettua määrää eri tyyppejä.

Käyttääksesi muita Python-tyyppejä sqlite3-tietokannoissa, sinun tulee adaptoida ne joksikin sqlite3-kirjaston tukemaksi tyypiksi, eli yhdeksi seuraavista:

NoneType	int	long	float	str	unicode	buffer

Olioiden adaptointiin käytettävä protokolla on nimeltään PrepareProtocol.

On kaksi tapaa saada sqlite3-kirjasto adaptoimaan oma Python-tyyppi joksikin tuetuista tyypeistä.

Ensimmäinen tapa: Anna oliosi adaptoida itsensä

Tämä on hyvä keino kun kirjoitat luokkasi itse. Oletetaan esimerkiksi, että luokkasi näyttää tältä:

```
class Point(object):
  def __init__(self, x, y):
    self.x, self.y = x, y
```

Päätät nyt tallentaa **Point**-olion yhtenä Sqliten sarakkeena. Ensin sinun on valittava yksi tuetuista tietotyypeistä esittämään luokkaa **Point**.

Käytämme tässä merkkijonotyyppiä **str** ja erotamme koordinaatit toisistaan puolipisteellä. Tämän jälkeen meidän täytyy luoda luokalle metodi **conform** parametreilla **self** ja **protocol**, jonka tehtävä on palauttaa muunnettu arvo. Protokolla-parametri saa arvon **PrepareProtocol**.

```
import sqlite3
class Point(object):
  def __init__(self, x, y):
    self.x, self.y = x, y
  def __conform__(self, protocol):
    if protocol is sqlite3.PrepareProtocol:
      return "%f;%f" % (self.x, self.y)
con = sqlite3.connect(":memory:")
cur = con.cursor()
p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

Toinen tapa: Kutsufunktion rekisteröinti adapteriksi

Toinen tapa saada sqlite3-kirjasto adaptoimaan oma Python-tyyppi joksikin tuetuista tyypeistä, on luoda funktio, joka muuntaa tyypin merkkijonoesitykseksi. Kyseinen funktio rekisteröidään metodilla **register_adapter**.

Adaptoitavan tyypin tai luokan tulee periytyä jotain reittiä object-luokasta.

```
import sqlite3
class Point(object):
  def __init__(self, x, y):
    self.x, self.y = x, y
def adapt_point(point):
  return "%f;%f" % (point.x, point.y)
sqlite3.register_adapter(Point, adapt_point)
con = sqlite3.connect(":memory:")
cur = con.cursor()
p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print cur.fetchone()[0]
```

Sqlite3-kirjastolla on kaksi oletusadapteria Pythonin vakiokirjaston olioiden **datetime.date** ja **datetime.datetime** tyyppejä varten. Oletetaan nyt, että haluamme tallentaa datetime.datetime olion, ei tällä kertaa ISO-standardin esityksenä, vaan Unixin aikaleimana.

```
import sqlite3import datetime, time
def adapt_datetime(ts):
  return time.mktime(ts.timetuple())
sqlite3.register_adapter(datetime.datetime, adapt_datetime)
con = sqlite3.connect(":memory:")
cur = con.cursor()
now = datetime.datetime.now()
cur.execute("select ?", (now,))
print cur.fetchone()[0]
```

Sqlite3-muuttujien muuntaminen omiksi Python-tyypeiksi

Adapterin luominen mahdollistaa omien Python-tyyppien lähettämisen Sqlitelle. Kuitenkin ollakseen käyttökelpoinen, menetelmä vaatii myös paluureitin Pythonista Sqliten kautta takaisin Pythonille.

Tällaiseen tehtävään käytetään muuntimia.

Palataan takaisin **Point**-luokan pariin. Tallensimme **(x,y)**-koordinaatit erotettuina puolipisteellä merkkijonoksi sqlite3-tietokantaan.

Määritellään ensin muunninfunktio, joka saa parametrinaan merkkijonon ja rakentaa siitä **Point**-luokan olion.

Huomaa, että muunninfunktioita kutsutaan aina merkkijonon avulla, olipa Sqlitelle lähetetty muuttuja minkä tietotyypin alaisuudessa hyvänsä.

```
def convert_point(s):
  x, y = map(float, s.split(";"))
  return Point(x, y)
```

Nyt on saatava sqlite3-kirjasto ymmärtämään, että valintamme tietokannasta onkin itse asiassa tyyppiä **Point** koordinaatteineen. Tämän toteuttamiseen on kaksi tapaa:

- Itseisarvoisesti määritellyn tyypin avulla
- Erikseen määritellyn sarakkeen nimen avulla

Molemmat tavat on kuvattuna kappaleessa Kirjaston funktiot ja attribuutit, vakioiden **PARSE_DECLTYPES** ja **PARSE_COLNAMES** määrittelyn yhteydessä.

Seuraava esimerkki valottaa molempia toteutustapoja:

```
import sqlite3
class Point(object):
  def __init__(self, x, y):
    self.x, self.y = x, y
  def __repr__(self):
    return "(%f;%f)" % (self.x, self.y)
def adapt_point(point):
  return "%f;%f" % (point.x, point.y)
def convert_point(s):
  x, y = map(float, s.split(";"))
  return Point(x, y)
# Rekisteroi adapteri
sqlite3.register_adapter(Point, adapt_point)
# Registeroi muunnin
sqlite3.register_converter("point", convert_point)
p = Point(4.0, -3.2)
#########################################
# 1) Maariteltyja tyyppeja kayttaen
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")
cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print "with declared types:", cur.fetchone()[0]
cur.close()
con.close()
#############################
# 2) Sarakkeen nimia kayttaen
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")
cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print "with column names:", cur.fetchone()[0]
cur.close()
con.close()
```

Oletusadapterit ja -muuntimet

Datetime-kirjaston tyypeille **date** ja **datetime** on olemassa oletusadapterit. Ne näkyvät ISO-standardin mukaisina päivämäärä- ja aikaleimoina sqlite3-tietokannalle.

Oletusarvoiset muuntimet rekisteröidään nimellä **date** olioille **datetide.date** ja nimellä **timestamp** olioille **datetime.datetime**.

Tällä tavoin voit käyttää päivämäärä- ja aikaleimoja Pythonista ongelmitta monissa tapauksissa. Adapterien muoto on yhteensopiva Sqliten päivä- ja aikafunktioiden kanssa. Seuraavassa esimerkki tästä:

```
import sqlite3
import datetime
con = sqlite3.connect(":memory:",
detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COCOLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")
today = datetime.date.today()
now = datetime.datetime.now()
cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])
cur.execute('select current_date as "d [date]", current_timestamp as "ts
[timestamp]"')
row = cur.fetchone()
print "Paivamaara nyt on", row[0], type(row[0])
print "Aikaleima nyt on", row[1], type(row[1])
```

Tiedonsiirtojen hallinta

Oletuksena sqlite3-kirjasto avaa yhteyden itseisarvoisesti ennen tiedonmäärityskielen lauseita, kuten INSERT, UPDATE, DELETE, REPLACE, ja toteuttaa muutokset itseisarvoisesti ennen muita kuin tiedonmäärityskielen lauseita ja hakuun liittymättömiä lauseita, eli mitä tahansa muita kuin SELECT, INSERT, UPDATE, DELETE, REPLACE.

Joten jos olet käsittelemässä avattua yhteyttä, ja suoritat komennon kuten CREATE TABLE, VACUUM, PRAGMA, sqlite3-kirjasto toteuttaa muutoksesi itseisarvoisesti ennenkuin suorittaa antamasi komennon. Tähän on kaksi syytä. Ensimmäinen on, että osa näistä komennoista ei toimi yhteyden aikana. Toinen syy on, että Sqliten on pidettävä kirjaa yhteyden tilasta, eli onko yhteys aktiivinen vaiko ei.

Voit hallita minkä tyyppisen BEGIN-lauseen Pysqlite oletusarvoisesti suorittaa, tai suorittaako mitään, käyttämällä eristystaso-parametria yhteyden **connect**-kutsulle, tai yhteyden eristystaso-ominaisuutta.

Jos haluat käyttöön automaattisesti suoritettavat toimenpiteet, käytä eristystasolle **isolation_level** arvoa **None**. Muussa tapauksessa jätä valinta oletusarvoonsa, jolloin tuloksena on puhdas BEGIN-lause, tai aseta se joksikin Sqliten tukemaksi eristystason arvoksi seuraavista: DEFERRED, IMMEDIATE tai EXCLUSIVE.

Lisätietoa: Sarakkeiden käsittely niiden nimien avulla järjestysnumeron käytön sijasta

Yksi sqlite3-kirjaston käyttökelpoinen ominaisuus on **row**-luokka, joka toimii rivin luontioliona.

Ne rivit, joita käytetään hyväksi tämän luokan avulla ovat kaikki saavutettavissa niin järjestysnumeron avulla kuten vektorit samoin kuin ei-merkkikokoherkkien nimien avulla.

```
import sqlite3
con = sqlite3.connect("mydb")
con.row_factory = sqlite3.Row
cur = con.cursor()
cur.execute("select name_last, age from people")
for row in cur:
  # Totea, etta...
  assert row[0] == row["name_last"]
  assert row["name_last"] == row["nAmE_lAsT"]
  assert row[1] == row["age"]
  assert row[1] == row["AgE"]
```

Toimenpiteen automaattinen loppuunsaattaminen

Toimenpide voidaan määrittää automaattisesti loppuunsaatettavaksi. Poikkeuksen sattuessa toimenpide perutaan, muutoin se suoritetaan.

```
from __future__ import with_statement
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar
unique)")
# Toimenpide onnistui, con.commit suoritetaan automaattisesti taman jalkeen
with con:
  con.execute("insert into person(firstname) values (?)", ("Joe",))
# Con.rollback-metodia kutsutaan kun with-osion suoritus paattyy poikkeukseen.
# Poikkeus nostetaan silti, ja se taytyy kasitella.
try:
  with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
  print "Ei voitu lisata Joea kahdesti"
```