Python-luento

Jonne Itkonen

April 11, 2012

Contents

1 It's...

Käynnistys ja sammutus komentoriviltä tapahtuu seuraavasti.

```
$ python
Python 2.5 (r25:51908, Jan 23 2007, 09:20:42)
[GCC 4.1.1 20070105 (Red Hat 4.1.1-51)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 'hello world'
'hello world'
>>> print 'hello world'
hello world
```

```
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
```

Yllä `$` on komentorivikehote (Unix-tyylinen, Windowsissa erilainen). Jokainen Python-tulkin kehote alkaa merkeillä `>>>`, joiden jälkeen on näytetty syöte. Voit kirjoittaa syötteen ilman kehoitemerkkejä omaan Python-tulkkiisi ja rivinvaihdon jälkeen sen pitäisi viimeisen kehoterivin jälkeen tulostaa esimerkissä näkyvä teksti.

Graafisempi versio Python-tulkista löytyy yleensä nimellä `idle`. Ohjelmia voi ajaa joko tuplaklikkaamalla niiden ikoneita tai antamalla ohjelmatiedosto parametrina `python`-komennolle.

Komentorivillä auttaa funktio `help`. Sen kutsuminen ilman argumenttia käynnistää avustuskehotteen. Argumentin kanssa kutsuttuna funktio tulostaa argumenttiin liittyvän avustuksen. (Kokeile antaa syöte `help()` Pythonin kehotteelle.)

Kutsumalla funktiota `quit()` voit poistua tulkista. Graafiset versiot toimivat tässä suhteessa kuten muutkin graafiset ohjelmat.

Nykyisin käytössäsi on todennäköisesti Pythonin versio 2.7 tai 3.0. Nämä esimerkit toimivat molemmissa versiossa joitain mainittuja poikkeuksia lukuunottamatta.

1.1 Laskentaa

```
>>> 1+1
2
>>> 1+2*3
7
>>> 5/2
2
>>> 5/2.
2.5
>>> 2**256
115792089237316195423570985008687907853269984665640564039457584007913129639936L
>>> 2.**256
1.157920892373162e+77
>>> (0+1j) * (0+1j)
(-1+0j)
>>>
```

Liukuluvut ovat liukulukuja Pythonissakin:

```
>>> 1.0 + 2.0
3.0
>>> 10.0 + 20.0
30.0
>>> 0.1 + 0.2
0.30000000000000004
>>> from decimal import Decimal
>>> Decimal('0.1')+Decimal('0.2')
```

```
Decimal('0.3')
>>>
```

Johtuen liukulukujen esitystavasta tietokoneessa, ne eivät koskaan ole tarkkoja lukuja. Pyöristysvirheitä syntyy helposti, kuten yllä nähdään. Tämän takia monista ohjelmointikielistä löytyy reaalilukujen aritmetiikkaa paremmin mallintavia tyyppejä, kuten Pythonin `Decimal`.

Tärkeintä on muistaa, että jo laskettaessa rahasummia voi ohjelmointikielen perusliukulukutyyppien käyttö aiheuttaa ikäviä yllätyksiä, joten rahamäärille kannattaa käyttää omaa luokkaansa.

1.2 Muuttujista

Muuttujat Pythonissa ovat käytännössä viitteitä. Ne antavat uuden nimen osoittamallensa oliolle, eivät varaa muistista tilaa, johon arvo sijoitetaan.

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> a = 2
>>> a
2
>>> a = 3.0
>>> a
3.0
>>> a = 'kissa istuu'
>>> a
'kissa istuu'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> a = 2
>>> a
2
>>> a = 3.0
>>> a
3.0
>>>
```

Jos kaksi muuttujaa osoittaa samaan *muuttumattomaan* olioon, ja toiseen sijoitetaan toinen olio, ei ensimmäinen muutu.

```
>>> a = 'kissa istuu'
>>> a
'kissa istuu'
>>> b = a
>>> b
'kissa istuu'
>>> b = 'istuuko kissa?'
```

```
>>> b
'istuuko kissa?'
>>> a
'kissa istuu'
>>>
```

Jos kaksi muuttujaa osoittaa samaan *muuttuvaan* olioon, ja toisen muuttujan kautta muutetaan oliota, näkyy muutos myös toisen muuttujan kautta.

```
>>> a=[1,2,3]
>>> b=a
>>> b[0]=2
>>> b
[2, 2, 3]
>>> a
[2, 2, 3]
>>>
```

Muuttujan sisältämän tai viittaaman arvon tyypin voi aina tarkistaa funktiolla `type()`. Tätä ei tule käyttää liikaa, se vain osoittaisi hutiloitua oliosuunnittelua.

```
>>> a=2
>>> type(a)
<type 'int'>
>>> a='kissa'
>>> type(a)
<type 'str'>
>>> type(2)
<type 'int'>
>>> type(type)
<type 'type'>
>>> type(Decimal)
<type 'type'>
>>> type(Decimal('1.0'))
<class 'decimal.Decimal'>
>>>
```

1.3 Muita tietotyyppejä

Listat muuttuvien ja järjestettyjen tietojoukkojen käsittelyyn:

```
>>> lista=[1, 2, 'kolme', 4]
>>> lista[0]
1
>>> lista[2]
'kolme'
>>> lista[-1]
4
>>> lista[1:3]
[2, 'kolme']
```

```
>>> lista[3:]
[4]
>>> lista[:2]
[1, 2]
>>> lista[-2]=3
>>> lista
[1, 2, 3, 4]
>>>
```

Monikot ovat kuten listat, mutta ne eivät voi muuttua: niiden koko pysyy samana, eikä edes alkioita voi muuttaa.

```
>>> monikko=(1, 2, 'kolme', 4)
>>> monikko
(1, 2, 'kolme', 4)
>>> monikko[0]
1
>>> monikko[-1]
4
>>> monikko[1:3]
(2, 'kolme')
>>> monikko[-2]=3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> len(lista)
4
>>> len(monikko)
4
>>> lista.append(5)
>>> lista
[1, 2, 3, 4, 5]
>>> len(lista)
5
>>> monikko.append(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> monikko
(1, 2, 'kolme', 4)
>>>
```

Joitain listaolioiden metodeja:

```
>>> lista.reverse()
>>> lista
[5, 4, 3, 2, 1]
>>> lista.sort()
>>> lista
[1, 2, 3, 4, 5]
>>> lista[-3]='kolme'
```

```
>>> lista
[1, 2, 'kolme', 4, 5]
>>> lista.sort()
>>> lista
[1, 2, 4, 5, 'kolme']
>>> lista[-1]=3
>>> lista.sort()
>>> lista.insert(0,0)
>>> lista
[0, 1, 2, 3, 4, 5]
>>> lista.pop()
5
>>> lista.append(5)
>>> lista
[0, 1, 2, 3, 4, 5]
>>> lista.extend([6,7])
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7]
>>> lista.pop()
7
>>> lista
[0, 1, 2, 3, 4, 5, 6]
>>> del lista[-1]
>>> lista
[0, 1, 2, 3, 4, 5]
>>> lista.append(0)
>>> lista
[0, 1, 2, 3, 4, 5, 0]
>>> lista.count(0)
2
>>> lista.remove(0)
>>> lista
[1, 2, 3, 4, 5, 0]
>>> lista.reverse()
>>> lista
[0, 5, 4, 3, 2, 1]
>>> lista.sort()
>>> lista
[0, 1, 2, 3, 4, 5]
>>> lista[-3]='kolme'
>>> lista
[0, 1, 2, 'kolme', 4, 5]
>>> lista.sort()
>>> lista
[0, 1, 2, 4, 5, 'kolme']
>>>
```

Hajautustaulut (sanasto, *dictionary*, *hash table*, *map*) ovat paljon käytetty tietorakenne. Avain, jonka perusteella arvo haetaan, voi olla lähes mitä tahansa tyyppiä (tyypille täytyy löytyä funktioiden `hash` ja `cmp` toteutus, tai vas-

taavien metodien toteutus).

```
>>> htaulu={}
>>> htaulu={'sata':100, 'tuhat':1000, 10:'kymmenen'}
>>> htaulu
{'sata': 100, 10: 'kymmenen', 'tuhat': 1000}
>>> htaulu[10]
'kymmenen'
>>> htaulu[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
>>> htaulu['sata']
100
>>> htaulu.keys()
['sata', 10, 'tuhat']
>>> htaulu.values()
[100, 'kymmenen', 1000]
>>> htaulu.items()
[('sata', 100), (10, 'kymmenen'), ('tuhat', 1000)]
>>> htaulu.has_key(1)
False
>>> htaulu[1]='yksi'
>>> htaulu
{'sata': 100, 10: 'kymmenen', 1: 'yksi', 'tuhat': 1000}
>>>
```

Joukot sisältävät alkiot ilman järjestystä ja moninkertoja.

```
>>> joukko = set([1,1,2,3,5,8])
>>> joukko
set([8, 1, 2, 3, 5])
>>> joukko2 = set([3,4,5])
>>> joukko.union(joukko2)
set([1, 2, 3, 4, 5, 8])
>>> joukko.intersection(joukko2)
set([3, 5])
>>> joukko.difference(joukko2)
set([8, 1, 2])
>>> joukko2.difference(joukko)
set([4])
>>> joukko.symmetric_difference(joukko2)
set([1, 2, 4, 8])
>>>
```

2 Rakenteiden määrittely

2.1 Funktion määrittely ja dokumentointi

```
>>> def summa(a,b):
```

```
...     "Palauttaa kahden argumenttinsa summan."
...     return a+b
...
>>> summa(2,3)
5
>>> summa(0.1,0.2)
0.30000000000000004
>>> summa(Decimal('0.1'),Decimal('0.2'))
Decimal('0.3')
>>> summa('kissa ','istuu')
'kissa istuu'
>>> help(summa)

Help on function summa in module __main__:

summa(a, b)
    Palauttaa kahden argumenttinsa summan.
(END)
[Paina Q-näppäintä palataksesi kehoitteeseen.]
>>> type(summa)
<type 'function'>
>>>
```

Funktioita voi myös antaa argumentteina toisille funktioille ja palauttaa funktioiden arvoina. Funktio-ohjelmoijat ovat siis melkein kuin kotonaan Pythonin parissa.

2.2 Ehto ja toistolauseet: if, while, for, break, continue

```
>>> if 2>3:
...     print 'oletko aivan varma?'
... else:
...     print 'näin on!'
...
näin on!
>>>

>>> a=1
>>> while a<5:
...     print '%d. Spam' % a
...     a = a + 1
...
1. Spam
2. Spam
3. Spam
4. Spam
>>>

>>> menu = ('Spam','Spam','Spam', 'Spam')
>>> for i in menu:
```

```
...     print i
...
Spam
Spam
Spam
Spam
>>>

>>> a=1
>>> while True:
...     print '%d. Spam' % a
...     if a>4: break
...     elif a==2:
...         print '...with Spam, please!'
...         a = a + 1
...     else:
...         a = a + 1
...
1. Spam
2. Spam
...with Spam, please!
3. Spam
4. Spam
5. Spam
>>>
```

2.3 Luokan määrittely ja dokumentointi, olion luonti

Vaihdetaan tässä vaiheessa hieman esitystapaa. Voit kirjoittaa allaolevat määrittelyt tulkin kehoitteella, mutta järkevämpää on käyttää erillistä editoria. Aiemmin mainittu `idle` tulee useimpien Python-asennusten mukana, mutta mikä tahansa Python-kielen tunnistava editori, siis useimmat editorit, on hyvä vaihtoehto. Itse käytän Emacsia ja sen Python-moodia. Tätä tehdessä olen käyttänyt Emacsin Org-moodia ja sen Babel-laajennosta. Vim, varsinkin Python-skriptauslaajennoksella, on myös oiva työväline. Laajennoksia löytyy myös muille, kuten Eclipselle, Netbeansille ja VisualStudiolle.

Rivinumeroita ei kirjoiteta, ne ovat vain olemassa, jotta voin viitata tekstissä jollekin tietylle riville esimerkissä.

Asiaan... Seuraavassa määritellään yksinkertainen henkilöluokka vanhalla ja uudella tavalla. Vanha tapa on vielä käytössä Pythonin versiossa 2, mutta Python 3 käyttää vain uutta tapaa. Version näet helpoiten Pythonin käynnistyksen yhteydessä, katso vaikka ihan ensimmäistä esimerkkiä tässä paperissa.

```
1: # -*- coding: iso-8859-15 -*-
2: # Tuo ylempi rivi kertoo, mitä merkistökoodausta lähdekoodissa käytetään.
3: # Kätevämpää kuin osaa ensin arvatakaan! Ääkkösilläkin on mahdollisuus
4: # tulostua oikein. Kannattaa käyttää samaa merkistökoodausta kuin
5: # tiedostojärjestelmässä, ja sen kannattaisi jo nykyään olla ainakin UTF-8.
6: # Python-tulkki kyllä murmuttaa, jos tuo ei ole kunnossa, ja neuvoo,
7: # mistä löytyy apu.
8: #
```

```
 9: # Jos tuntui liian nörtiltä , kopioi vain ensimmäinen rivi tai älä käytä
10: # ääkkösiä :)
11:
12: class Person:
13:     """Person is a class for objects holding a persons name and phone number."""
14:
15:     def __init__(self, name, phone):
16:         self._name = name
17:         self._phone = phone
18:
19:     def name(self):
20:         return self._name
21:     def set_name(self, name):
22:         self._name = name
23:         return self._name
24:
25:     def phone(self, phone=None):
26:         """This style of attribute accessor combines
27:         the setter and getter in one method. Call it
28:         without arguments to get the value, and with
29:         single argument to set the value.
30:
31:         This is also an example of multi-line documentation string :)
32:         Try help(Person) or help(Person.phone) to see the documentation.
33:         """
34:         if phone:
35:             self._phone = phone
36:         return self._phone
37:
38:     def __str__(self):
39:         return 'Person name: %s phone: %s' % (self.name(), self.phone())
40:
41: class NewStylePerson(object): # New style class definition
42:     def __init__(self, name):
43:         self._name = name
44:
45:     def get_name(self):
46:         return self._name
47:     def set_name(self, name):
48:         self._name = name
49:         return self._name
50:     name = property(get_name, set_name)
51:
52:     # Property phone done likewise , left out for clarity .
53:
54:     def __str__(self):
55:         return 'Person name: %s' % self.name
56:
57:
58: donald = Person('Donald','555-313')
59: daisy = NewStylePerson('Daisy')
60: print donald
61: print daisy
62: print Person.__doc__
63: # Remember to try help(Person)
```

```
Person name: Donald phone: 555-313
Person name: Daisy
Person is a class for objects holding a persons name and phone number.
```

Vanhan ja uuden ero on, että uuden mallisen luokkamäärittelyn tulee

mainita yliluokkana `object` perintälistassa, eli suluissa uuden luokan nimen perässä, kts. `NewStylePerson` rivillä 41. Metodien ensimmäinen parametri tulee olla `self`, eli viite olioon itseensä. Lohkot merkitään sisennyksellä kuten aiemminkin. Metodi `__init__()` toimii rakentimena, ja kaikki olion attribuutit onkin määriteltävä siellä. Jos määrittelet attribuutin metodin ulkopuolella, siitä tulee *luokan* attribuutti.

Metodi `__str__` vastaa Javan `toString()`-metodia, eli sen pitää palauttaa tulostamiskelpoinen ja ihmiselle lukukelpoinen merkkijono, joka kertoo tarpeelliset asiat oliosta.

Olio luodaan yksinkertaisesti antamalla luokan nimen perässä suluilla ympäröitynä olion rakentimen argumentit. Sulun ja luokan nimen välissä ei ole välilyöntiä. Palautuva olioviite tallennetaan muuttujaan normaalilla sijoituslauseella.

2.4 Poikkeuksellisesti: try ... except

Tulossa myöhemmin.

3 Testilähtöinen ohjelmistokehitys

```
 1: # −*− coding: iso−8859−15 −*−
 2: import unittest
 3: from money import Money
 4:
 5: class MoneyTest(unittest.TestCase):
 6:     def setUp(self):
 7:         self.m12CHF = Money(12, 'CHF')
 8:         self.m14CHF = Money(14, 'CHF')
 9:         self.m28USD = Money(28, 'USD')
10:     def testSimpleAdd(self):
11:         expected = Money(26, 'CHF')
12:         result = self.m12CHF.add(self.m14CHF)
13:         self.assertEquals(expected, result, 'not expected value')
14:
15: if __name__=='__main__':
16:     unittest.main()
```

Yllä on testilähtöisen kehityksen perinne-esimerkistä pätkä kirjoitettuna Pythonilla. Koetapa toteuttaa `Money`-luokka tuon avulla! Seuraava tehtävä: Tee keilailu-kata Pythonilla: http://butunclebob.com/files/downloads/Bowling%20Game%20Kata.ppt .

4 Moduulit ja paketit

```
1: # −*− coding: iso−8859−15 −*−
2:
3: # tiedosto: kala.py
4:
5: a=175
6:
7: def foo():
8:     print 'foo!'
```

```
 9:
10:  def bar():
11:      print 'bar!'
12:
13:  #
14:  bar()
15:
16:  if __name__=='__main__': # aja seuraavaa import−lauseessa
17:      foo()
```

```
bar!
foo!
```

Jos halutaan rajoittaa osa tiedoston suorituksesta vain siihen, kun tiedosto ajetaan Pythonilla, se tehdään rivillä 16 alkavalla `if`-lohkolla.

Jos tämä puuttuu, ja tilalla on pelkkä unittest.main(), ajetaan tuo unittest.main() joka kerta kun tiedosto luetaan, eli esimerkiksi joka kerta kun tiedosto otetaan mukaan toiseen import komennolla.

Funktio `dir` listaa argumenttinsa attribuutit. Muista myös funktio `help`.

```
>>> import kala
>>> dir(kala)
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'foo']
>>> dir()
['__builtins__', '__doc__', '__name__', 'kala']
>>>
```

Moduulin attribuuttien tuominen osaksi käytössä olevaa nimiavaruutta ei yleensä ole järkevää, vaan kuormittaa nimiavaruuden turhilla määrityksillä:

```
>>> from kala import *
>>> dir(kala)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'kala' is not defined
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'foo', 'os']
>>>
```

Tuo vain tarpeellinen, tai sitten koko moduuli.

```
>>> from kala import foo
>>> dir(kala)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'kala' is not defined
>>> dir()
['__builtins__', '__doc__', '__name__', 'foo']
>>>
```

Hätätapauksessa voit uudelleennimetä tuomasi kohteen, mutta tätä on syytä käyttää harkiten.

```
>>> from kala import foo as bar
>>> dir(kala)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'kala' is not defined
>>> dir()
['__builtins__', '__doc__', '__name__', 'bar']
>>> bar()
foo!
>>>
```

4.1 Paketit

Paketit toteutetaan hakemistohierarkiana. Esimerkiksi paketti `a.b.c.foo` löytyy hakemistopolusta `a/b/c/foo.py`. Jokaiseen alihakemistoon tulee laittaa tiedosto `__init__.py`, joka voi olla tyhjäkin. Tiedostoon voi kuitenkin kirjoittaa paketin alustuskoodin, joka ajetaan, kun paketin sisältö otetaan käyttöön (*import*). Tiedostossa voi myös määritellä attribuutin `__ALL__`, joka kertoo, mitä paketista otetaan käyttöön, jos käytetään muotoa `import * from paketti`.

4.2 Esimerkkejä kirjastoista

Käydään läpi esimerkkejä Pythonin varsin kattavasta oheiskirjastosta. Leikkisästi voidaan sanoa, että jos jotain ei Pythonille löydy, sitä ei tarvita. (Eli kannattaa etsiä ennen kuin itse alkaa koodaamaan...)

5 Tieto talteen

```
>>> f=open('/tmp/kala.txt','wt')
>>> f.write('kissa istuu puussa\n')
>>> print >>f, 'jos kuu on juustoa, niin minä...'
>>> f.close()

$ cat /tmp/kala.txt
kissa istuu puussa
jos kuu on juustoa, niin minä...
$

>>> l=('kissa istuu puussa\n','jos kuu on...\n')
>>> open('/tmp/kala-b.txt','wt').writelines(l)

$ cat /tmp/b
kissa istuu puussa
jos kuu on...
$

>>> f=open('/tmp/kala.txt','rt')
>>> f.read()
'kissa istuu puussa\njos kuu on juustoa, niin minä...\n'
```

```
>>> f.seek(0)
>>> f.readline()
'kissa istuu puussa\n'
>>> f.readline()
'jos kuu on juustoa, niin minä...\n'
>>> f.readline()
''
>>> f.read()
''
>>> f.seek(0)
>>> for rivi in f:
...     print rivi
...
kissa istuu puussa

jos kuu on juustoa, niin minä...

>>> f.close()
>>> for rivi in open('/tmp/kala.txt','rt'):
...     print rivi
...
kissa istuu puussa

jos kuu on juustoa, niin minä...

>>> for rivi in open('/tmp/kala.txt'):
...     print rivi
...
kissa istuu puussa

jos kuu on juustoa, niin minä...

>>>

>>> from __future__ import with_statement
>>> with open('/tmp/kala.txt') as f:
...     for rivi in f:
...             print rivi
...
kissa istuu puussa

jos kuu on juustoa, niin minä...

>>>
```

5.1 Tiedon sarjallistaminen

```
>>> class foo:
...     def __init__(self, x):
...             self.__x = x
```

```
...     def __repr__(self):
...             return '<foo@%x arvolla %d.>' % (id(self), self.__x)
...
>>> data=(1, 2.0, 'kissa', [4, 5, 6], foo(16))
>>> data
(1, 2.0, 'kissa', [4, 5, 6], <foo@2aaaae4c7e60 arvolla 16.>)
>>> import pickle
>>> pickle.dump(data, open('/tmp/dumppi','w'))
>>> pickle.load(open('/tmp/dumppi'))
(1, 2.0, 'kissa', [4, 5, 6], <foo@2aaaaeb51a70 arvolla 16.>)
>>>
```

5.2 Liitynnät tietokantoihin

Seuraavat esimerkit ovat (lähes) suoraan Pythonin `sqlite3`-moduulin dokumentaatiosta.

```
1:  import sqlite3
2:
3:  conn = sqlite3.connect('/tmp/example')
4:  c = conn.cursor()
5:
6:  # Create table
7:  c.execute('''create table stocks
8:  (date text, trans text, symbol text,
9:   qty real, price real)''')
10:
11: # Insert a row of data
12: c.execute("""insert into stocks
13:           values ('2006-01-05','BUY','RHAT',100,35.14)""")
14:
15:
16: ## *** NEVER do this -- INSECURE!!!! ***
17: ## symbol = 'IBM'
18: ## c.execute ("... where symbol = '%s'" % symbol)
19:
20: # Do this instead
21: t = (symbol,)
22: c.execute('select * from stocks where symbol=?', t)
23:
24: # Larger example
25: for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
26:           ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
27:           ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
28:          ):
29:     c.execute('insert into stocks values (?,?,?,?,?)', t)
```

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

Lähde: http://docs.python.org/lib/module-sqlite3.html

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print "Enter your SQL commands to execute in sqlite3."
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()
```

Lähde: http://docs.python.org/lib/sqlite3-Module-Contents.html

Yleisesti Pythonille tehtyjen tietokantaliityntöjen tulisi toteuttaa API, joka on esitelty osoitteessa http://www.python.org/dev/peps/pep-0249/. Yllä esitelty `sqlite3`-moduuli on yksi esimerkki Python-DB-rajapinnan toteutuksesta.

Pythonille tietokantaa etsiessä kannattaa myös vilkaista ZOPE-sovelluspalvelimen[1] tietokantaa ZODB[2]. ZODB on oliotietokanta, joten ainakin puhtaat oliosovellukset hyötyvät sen käytöstä. Seuraava esimerkki on ZODB:n ohjeesta osoitteesta http://wiki.zope.org/ZODB/guide/node3.html.

```
from ZODB import FileStorage, DB

storage = FileStorage.FileStorage('/tmp/test-filestorage.fs')
db = DB(storage)
conn = db.open()

from persistent import Persistent

class User(Persistent):
    pass
```

[1] http://www.zope.org/ Pythonilla tehty WWW-sovelluspalvelin.
[2] http://wiki.zope.org/ZODB/FrontPage Tietokantatoteutus ZOPE:n alla.

```
11:
12:  dbroot = conn.root()
13:
14:  # Ensure that a 'userdb' key is present
15:  # in the root
16:  if not dbroot.has_key('userdb'):
17:      from BTrees.OOBTree import OOBTree
18:      dbroot['userdb'] = OOBTree()
19:
20:  userdb = dbroot['userdb']
21:
22:  # Create new User instance
23:  import transaction
24:
25:  newuser = User()
26:
27:  # Add whatever attributes you want to track
28:  newuser.id = 'amk'
29:  newuser.first_name = 'Andrew' ; newuser.last_name = 'Kuchling'
30:   ...
31:
32:  # Add object to the BTree, keyed on the ID
33:  userdb[newuser.id] = newuser
34:
35:  # Commit the change
36:  transaction.commit()
```

```
>>> newuser
<User instance at 81b1f40>
>>> newuser.first_name           # Print initial value
'Andrew'
>>> newuser.first_name = 'Bob'   # Change first name
>>> newuser.first_name           # Verify the change
'Bob'
>>> transaction.abort()          # Abort transaction
>>> newuser.first_name           # The value has changed back
'Andrew'
>>>
```