Python

Kurssin lopputyö

Kristian Ovaska

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos
Ohjelmointikielten periaatteet 2004
Helsinki 29. huhtikuuta 2004

Sisältö

1 Yleiskatsaus

Python on oliopohjainen skriptikieli, joka sopii myös laajempien ohjelmistojen toteuttamiseen. Kielen piirteitä ovat dynaaminen tyypitys, pelkistetty syntaksi ja monipuoliset sisäänrakennetut tietotyypit sekä standardikirjasto. Python on moniparadigmakieli: oliopohjaisuuden lisäksi se sopii proseduraaliseen ja funktionaaliseen ohjelmointiin. Dynaamisen luonteensa vuoksi Python sopii parhaiten tulkattavaksi kieleksi; konekoodia tuottavaa Python-kääntäjää ei toistaiseksi ole olemassa.

Kirjoituksessa käydään läpi Pythonin ominaisuuksia ja historiaa. Joidenkin ominaisuuksien kohdalla mainitaan, mistä kielestä Python on saanut vaikutteita. Tällöin tyypillisesti rajataan tarkastelu välittömiin vaikuttajiin eikä seurata historiaa pidemmälle.

Merkeillä >>> alkavat koodiesimerkit kuvaavat suoritusta interaktiivisessa tulkissa.

1.1 Historia

Pythonin kehitti hollantilainen Guido van Rossum 80-luvun lopussa ja 90-luvun alussa [Ven03].

Kielen edeltäjä oli opetuskäyttöön suunniteltu kieli ABC [Mee84], jonka parissa Van Rossum työskenteli 1983-86/87. ABC-kielen työnimi oli B, joka sittemmin muutettiin ABC:ksi [Mee92, s. 2]; kyseinen B on eri kieli kuin Ken Thompsonin B-kieli, joka oli C-kielen taustalla [Ker88, s. 1].

ABC:stä Van Rossum otti Pythoniin hyvänä pitämiään ominaisuuksia, kuten sisennyksen käyttäminen ohjelmalohkojen merkitään [Kuc98], tärkeimmät tietorakenteet (listat ja hajautustaulukot), interaktiivinen tulkki ja for-silmukan toiminta.

80-luvun lopussa Van Rossum työskenteli Amoeba-nimisen hajautetun käyttöjärjestelmän parissa ja kehitti Pythonia alunperin sen skriptikieleksi. Kie-

len toteuttaminen alkoi loppuvuodesta 1989 [FAQ]. Ensimmäinen yleiseen käyttöön tarkoitettu versio 0.9 julkaistiin helmikuussa 1991 [Ros91].

Python 1.0 julkaistiin vuonna 1994 ja Python 2.0 vuonna 2000. Tämä kirjoitus käsittelee versiota 2.3, joka on toistaiseksi kielen uusin versio.

1.2 Kielen toteutuksia

Pythonin alkuperäinen ja de-facto -toteutus on CPython [Pyt], jota kutsutaan usein vain Pythoniksi. CPython on C-kielellä toteutettu tavukooditulkki.

Jython on Java-kielinen toteutus, joka kääntää Python-koodia Javan tavukoodiksi ja mahdollistaa Python-ohjelmien ajamisen Javan virtuaalikoneessa.

Psyco on laajennusmoduuli CPythoniin, joka kääntää CPythonin tavukoodia ajonaikaisesti x86-konekoodiksi (just-in-time -käännös).

PyPy [PyPy] (Python in Python) on tutkimushanke, joka tähtää Python-toteutuksen luomiseen kielellä itsellään. Tavoitteena on CPythonia tehokkaampi Python-koodin suorittaminen.

1.3 Kehitysprosessi

Pythonia kehitetään avoimen lähdekoodin projektina. Guido van Rossum päättää viime kädessä kaikesta (C)Pythoniin liittyvästä, mutta kuka tahansa voi osallistua kehitystyöhön. Merkittävät muutosehdotukset vaativat yleensä Python Enhancement Proposal -dokumentin [PEP1], joka avulla Van Rossum ja muu ydinryhmä arvioi ehdotusta.

2 Tietoabstraktiot

Tässä luvussa käsitellään kahta kapselointimenetelmää: olioita ja moduuleita.

2.1 Oliot

Olio on keskeinen rakenne Pythonissa [Ros03a, k. 3.1]. Kaikki suoritusaikaiset rakenteet, kuten luokat, luokkien ilmentymät, funktiot ja lukuarvot ovat olioita. Kaikilla olioilla ei kuitenkaan ole luokkaa, vaan olio saattaa olla jonkin sisäänrakennetun tyypin ilmentymä. Tyypin ja luokan raja on häilyvä: monet tyypit käyttäytyvät kuin luokat ja niitä voi periä omiin luokkiin. Esimerkiksi kokonaislukutyyppiä voi käyttää ”kantatyyppinä”, mutta funktiotyyppi ei tätä kuitenkaan salli.

Luokkamäärittelyissä metodien ja attribuuttien nimet noudattavat seuraavia konventioita: `_x` vastaa jossain määrin protected-määrettä, `__x` private-määrettä ja `__x__` merkitsee erikoismetodia, joka kuuluu kielen määrittelyyn [Ros03a, k. 3.3]; esimerkiksi `__init__` on luokan alustaja. Operaattoreita voi kuormittaa erikoismetodien avulla: esimerkiksi `__add__` kuormittaa yhteenlaskuoperaattorin.

Python ei ole ”puhdas” oliokieli. Funktiot voi sijoittaa moduulin päätasolle ja käyttää kieltä proseduraaliseen tapaan. Standardikirjastossa [Ros03b] on paljon tällaisia moduuleita. Luokkien attribuutit ovat oletuksena julkisia, ja yksityisten attribuuttien (ja metodien) käyttäminen on vaivalloista, koska on joka kerta kirjoitettava nimeen kuuluvat alaviivat (`__x`). ”Protected-määre” on pelkkä nimeämiskonventio eikä kuulu varsinaisesti kieleen.

Pienenä erikoisuutena metodien formaalissa parametrilistassa on ensimmäisenä parametrina ilmentymä, jonka kautta metodia kutsutaan:

```
class Luokka:
    def metodi(self, x): pass
```

Tämä on Van Rossumin mukaan [Kuc98] peräisin Modula-2/3:sta, jossa proseduuri saa usein parametriksi moduulin hallinnoiman tyypin arvon [Sco00, s. 126]. `self`-parametri vastaa Java-kielen avainsanaa `this`.
Pythonissa `ilmentyma.metodi(x)` ja `Luokka.metodi(ilmentyma, x)` tuottavat saman kutsun.

2.2 Moduulit

Olioiden ohella keskeinen rakenne on moduuli. Python-kielinen moduuli voi sisältää luokkien ja funktioiden määrittelyitä sekä sellaisenaan suoritettavaa koodia, ja C-kielellä toteutettu moduuli mahdollistaa kielen yhdenmukaisen laajentamisen [Ros03c].

Moduulijärjestelmä on saanut vaikutteita Modula-2/3:sta. Esimerkiksi import-lause on hyvin samanlainen [Car89, s. 33].

3 Tyypitys

Python on dynaamisesti ja melko vahvasti tyypitetty. Nimensidontamalli (kts. luku 4) sallii mielivaltaisen olion sitomisen nimeen ajoaikana, joten staattista tarkistusta ei voida tehdä. Ajoaikana tehdään tyyppitarkistuksia: esimerkiksi `1+'2'` , `5[0]` ja `x=5; x()` aiheuttavat poikkeuksen.

Funktio- ja metodikutsuissa sovelletaan rakenne-ekvivalenssia: kutsu `obj.met(5)` onnistuu, jos oliolla `obj` on metodi `met`, jota voi kutsua yhdellä argumentilla (jos argumentti on väärän tyyppinen, metodin suoritus voi tietysti aiheuttaa poikkeuksen). Periaatteessa esim. funktion argumenttina saatu `obj` voi olla vääränlainen olio, jolla vain sattuu olemaan metodi `met`.

Tyypit jaetaan muokattaviin (mutable) ja muokkaamattomiin (immutable). Muokkaamattomien tyyppien arvoa ei voi muuttaa olion luomisen jälkeen; näin ne sopivat hajautustaulun avaimiksi (kts. kappale 3.2).

Seuraavissa kappaleissa käsitellään yleisimpiä sisäänrakennettuja tyyppejä [Ros03a, k. 3.2], joita ohjelmoija käyttää suoraan. Pythonissa kaikilla ajonaikaisilla olioilla on tyyppi, joten on myös moduulityyppi, funktiotyyppi ym. Niitä ei käsitellä.

3.1 Alkeistyypit

Lukutyyppejä ovat kiinteän- (int) ja mielivaltaisen (long) mittaiset kokonaisluvut, liukuluvut ja kompleksiluvut. Totuusarvo (bool) on teknisesti inttyypin alityyppi ja sitä käytetään totuusarvovakioiden (True ja False) ilmaisemiseen. bool-tyyppi on varsin uusi ominaisuus kielessä; ennen totuusarvovakiot ilmaistiin C-kielen kaltaisesti [Ker88, s. 42] kokonaisluvuilla.

NoneType-tyyppi ja sen singleton-ilmentymä None vastaa muiden kielten null- tai nil-arvoa.

Atomista merkkityyppiä ei ole; yksittäinen merkki ilmaistaan merkkijonona. Kaikki alkeistyypit ovat muokkaamattomia.

3.2 Yhdistelmätyypit

Yhdistelmätyypit voidaan jakaa kuvaus- (map) ja sekvenssityyppeihin (sequence).

Ainoa kuvaustyyppi on hajautustaulukko (dict[ionary]). Se kuvaa avainolion mielivaltaiseksi arvo-olioksi. Avaimen on oltava muokkaamaton, koska avain etsitään hajautusarvon (hash) avulla, joka ei saa muuttua avainolion elinkaaren aikana. Hajautustaulukko on sisäisesti tärkeä tyyppi, koska nimiavaruudet toteutetaan sen avulla (kts. luku 4).

Sekvenssityyppejä ovat listat (list), monikot (tuple) ja merkkijonot (str ja unicode). Listat ja monikot sisältävät järjestetyn kokoelman olioita. Listat on toteutettu muuttuvanmittaisina taulukoina, ei siis Lispin [McC60] kaltaisina linkitettyinä listoina. Monikot ovat muuten listojen kaltaisia, mutta muokkaamattomia. Tämän vuoksi ne sopivat hajautustaulun avaimiksi, tosin vain, jos myös kaikki monikon alkiot ovat muokkaamattomia.

Sekvensseistä voi luoda uusia sekvenssejä viipaloimalla (slicing). Syntaksi on saanut vaikutteita Algol-68:sta [Tan76, s. 170], jossa sitä kutsutaan termillä trimming, sekä Iconista [Gri96, s. 56]. Esimerkki viipaloinnista:

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

3.2.1 Listakomprehensiot

Yksi tapa tuottaa listoja on listakomprehensiot (list comprehensions) [PEP202], jotka tuottavat listan alkiot for-silmukassa toistettavan lausekkeen ja valinnaisen if-lauseen avulla:

```
>>> [x**2 for x in [0, 1, 2, 3, 4] if x>=2]
[4, 9, 16]
```

Piirre on samankaltainen kuin Haskellissa [Jon03, k. 3.11], jossa em. lista generoidaan lausekkeella `[x**2 | x <- [0, 1, 2, 3, 4], x>=2]`, ja piirteen juuret ovat matemaattisessa joukkomerkinnässä $\{x^2 \mid \forall x \in [0,4] : x >= 2\}$.

4 Nimien hallinta

Yhtenäisen oliomallin vuoksi on luontevaa, että muuttujat noudattavat Pythonissa viitesemantiikkaa: muuttuja on nimi, jotka viittaa (eli on sidottu) johonkin olioon [Ros03a, k. 4.1]. Nimen voi sitoa mihin tahansa olioon, joten nimillä ei ole tyyppiä, vain olioilla on.

Nimiavaruus (namespace) on Pythonissa rakenne, joka sisältää nimisidonnat. Moduuleilla, luokilla, ilmentymillä ja funktiokutsuilla on ajoaikana omat nimiavaruutensa, jotka on sisäisesti toteutettu hajatustaulukkoina. Nimen etsintä etenee hierarkkiassa ylöspäin, esimerkiksi funktiosta moduuliin ja sisäänrakennettuun nimiavaruuteen.

Hieman epäortogonaalisena piirteenä metodista ei näe suoraan luokan nimiavaruuteen, vaan viittaukset metodeihin ja staattisiin luokkamuuttujiin

on tehtävä self-parametrin kautta. Ilmentymämuuttujiin viitataan samalla tavalla.

Funktiokutsujen yhteydessä sovelletaan leksikaalista skooppia. Luokalla ja ilmentymällä on eri nimiavaruudet. Luokan nimiavaruus sisältää metodit ja luokkamuuttujat, ilmentymän nimiavaruus ilmentymämuuttujat.

Nimiavaruus muuttuu olemassaoloaikanaan, kuten seuraavasta käy ilmi (`locals` on funktio, joka löytyy sisäänrakennetusta nimiavaruudesta):

```
def f(a, b):
    print locals()          # Tulostaa {'a': 1, 'b': 2}
    z = 5; print locals()   # Tulostaa {'a': 1, 'b': 2, 'z': 5}
    del b; print locals()   # Tulostaa {'a': 1, 'z': 5}
f(1, 2)
```

5 Ohjausrakenteet

Tässä luvussa käsitellään joitakin Pythonin ohjausrakenteita.

5.1 Funktiot

Funktiot ja metodit määritellään def-lauseella. def on ajoaikana suorittettava lause samoin kuin if tai while. def-lauseen suorittaminen luo funktio-olion ja sitoo sen ympäröivään nimiavaruuteen. Myös mahdolliset parametrien oletusarvot evaluoidaan tässä vaiheessa. Funktion (tavu)koodi on käännetty etukäteen.

Koska funktiot ovat yhtenäisen oliomallin ansiosta ensiluokan olioita (first-class objects), sopii Python tältä osin funktionaaliseen ohjelmointiin. Myös listakomprehensiot tukevat funktionaalista mallia. Toisaalta Python ei muuta häntärekursiota iteratiivikseksi rakenteeksi, mikä yleensä on funktionaalisten kielten piirre.

Pythonissa on funktionaalisista kielistä tutut lambda-lausekkeet, jotka tuottavat nimettömän funktion, jonka voi suorittaa halutuilla parametreilla. Lambda-rakenne on Pythonissa rajoitettu: se voi sisältää vain lausekkeita, ei lauseita kuten if- ja for-lauseet. Lambda-avainsana on peräisin lambda-kalkyylistä; ohjelmointikielissä sen esitteli Lisp [McC60, s. 189].

5.2 Iteraatio ja generaattorit

Iteraatio tapahtuu for-lauseella: `for x in kokoelma`. Iteroitavan olion on tuettava Pythonin iteraatioprotokollaa [PEP234], joka mahdollistaa yhtenäisen iteraation erilaisille rakenteille. Sisäänrakennetut sekvenssi- ja kuvaustyypit tukevat protokollaa, samoin tiedostotyyppi file, joka mahdollistaa tekstitiedoston rivien iteroimisen. Omiin luokkiin voi rakentaa tuen iteraatiolle.

Seuraava esimerkki havainnollistaa iteraatioprotokollan toimintaa:

```
>>> lista = [1, 2]
>>> iter = lista.__iter__()
>>> type(iter)
<type 'listiterator'>
>>> iter.next()
1
>>> iter.next()
2
>>> iter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Kokoelmatyypin metodi `__iter__` palauttaa iteraattorin, jolla on metodi `next`. Kun iteraatio on käyty loppuun, iteraattori nostaa `StopIteration`-poikkeuksen. Käytännössä iteraatio tehdään yleensä for-silmukan avulla, joka kutsuu automaattisesti `__iter__`-metodia ja käsittelee poikkeuksen.

Yksi keino toteuttaa iteraattori on generaattorifunktio [PEP255], joka muistuttaa tavallista funktiota, mutta return-lauseen sijasta käyttää yield-lausetta. Generaattorit ovat saaneet vaikutteita CLU:sta [Lis81, s. 70], joka myös käyttää yield-avainsanaa ja samankaltaista for-lausetta iterointiin.

Seuraava esimerkki havainnollistaa generaattorifunktion toimintaa:

```
>>> def gen():
...     yield 1; yield 2
>>> geniter = gen()
>>> geniter2 = gen()
>>> geniter.next()
1
>>> geniter.next()
2
>>> geniter2.next()          # kutsutaan välillä toista
1
>>> geniter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Kutsu `gen()` tuottaa ilmentymän (generaattori-iteraattorin) generaattorifunktiosta. Samasta generaattorifunktiosta voi tuottaa monta itsenäistä iteraattoria. Metodikutsu `next()` siirtää suorituksen funktion runkoon, josta palataan yield-lauseella. Seuraava `next()`-kutsu jatkaa siitä kohdasta, mihin edellinen jäi. Kun generaattorin runko on käyty loppuun, nostetaan poikkeus.

5.3 Poikkeukset

Python käyttää poikkeuksia virheiden hallitsemiseen ja myös iteraatioprotokollassa ilmoittamaan iteraation päättymisestä. Standardikirjastossa on poikkeushierarkia, josta sovellusohjelmat voivat periä omia poikkeusluokkia.

Poikkeusmekanismi on Van Rossumin mukaan [Kuc98] saanut vaikutteita Modula-2/3:sta. Avainsanat (raise, try..except, try..finally) ovat kielissä samat [Car89, s. 21].

6 Esimerkkiohjelma

Tässä luvussa käydään läpi esimerkkiohjelma, jonka lähdekoodi on liitteessä A. Ohjelma tekee lukuarvoille yksikkömuunnoksia, esim. senttimetreistä tuumiksi. Lähdekoodissa on selvyyden vuoksi rivinumerot.

Riveillä 3-12 määritellään hajautustaulukko, jossa on tiedot yksiköiden muuntamisesta. Taulukon avaimena on pari (lähtöyksikkö, kohdeyksikkö) ja arvona joko muuntokerroin (esim. 1 euro = 5,94573 mk) tai yksiparametrinen lambda-lauseke, jota käytetään Celcius- ja Fahrenheit-muunnoksissa, koska asteikkojen välillä ei voi muuntaa yksinkertaisella kertoimella. Hajautustaulukon nimi (`CONVERSION_DICT`) kirjoitetaan isolla merkitsemään vakiorakennetta. Kyseessä on pelkkä konventio; kieli ei estä rakenteen muokkaamista.

Riveillä 14-31 määritellään funktio `convert`, joka muuntaa arvon yksiköstä toiseen. Funktion ensimmäinen lause on merkkijonoliteraali, jonka kieli tulkitsee funktion dokumentaatioksi. Se on ajoaikana saatavilla nimellä `convert.__doc__`.

Riveillä 17-20 tarkistetaan argumentin `value` tyyppi lausekkeella, jossa testataan, että arvoon voidaan lisätä luku 1,5 ja arvo voidaan muuttaa float-tyyppiseksi. Jos operaatiot onnistuvat, oletetaan, että argumentti on kelvollista tyyppiä. Lausekkeen tulosta ei sijoiteta mihinkään; semantiikan kannalta on tärkeää, ettei toteutus optimoi pois ”turhaa” lauseketta. Tarkistus voitaisiin tehdä testaamalla, onko luku tyyppiä int, long tai float, mutta tällöin rajoitutaan näihin lukutyyppeihin ja Pythoniin jatkossa lisättävät tai laajennusten määrittelemät lukutyypit voivat jäädä testin ulkopuolelle. Mahdollisia lukutyyppejä ovat esim. desimaali-, rationaali- ja kiinteän pilkun (fixed point) luvut. Rivillä 20 nostetaan poikkeus uudelleen kuvaavalla virheilmoi-

tuksella.

Riviltä 22 alkaa for-silmukka, joka iteroi yksikkömuunnostaulukon avainten ja arvojen yli. Hajautustaulukon metodi `iteritems` palauttaa iteraattorin, joka tuottaa (avain, arvo)-pareja. Rivillä 24 tarkistetaan arvon `factor` tyyppi sisäänrakennetulla funktiolla `type`. Jos arvo on lambda-lauseke, se evaluoidaan antamalla parametriksi numeerinen arvo, jolloin saadaan muunnettu luku. Muussa tapauksessa tulos saadaan kertolaskulla. Rivillä 28 tulostetaan muunnoksen tulos käyttäen %-operaattorilla tehtävää merkkijonojen interpolointia, joka on lainattu C-kielen printf-syntaksista.

Riveillä 30-31 on Pythonin pieni erikoisuus: for-silmukan else-haara, joka suoritetaan, jos silmukka käydään loppuun suorittamatta break-lausetta.

Ohjelman loppuosassa on komentorivikäyttöliittymän toteutus. Pythonissa ei ole C-kielen kaltaista `main`-erikoisfunktiota, mutta moduulin attribuutti `__name__` sisältää arvon `'__main__'` kun moduuli ajetaan pääohjelmana esim. komennolla `python unitconv.py`. Jos yksikkömuunnosmoduuli tuodaan toiseen moduuliin import-lauseella, on attribuutin `__name__` arvo moduulin nimi (esim. `'unitconv'`) eikä rivejä 34-38 suoriteta.

Rivillä 34 tehdään komentoriviargumenttilistasta viipaloitu kopio, jossa ensimmäinen elementti on jätetty pois. (`sys.argv[0]` on ajettavan ohjelman nimi ja varsinaiset argumentit alkavat indeksistä 1.) Viipaloinnin `[1:]` päätepiste on jätetty merkitsemättä, jolloin päätepisteeksi tulkitaan merkkijonon loppu.

Lähteet

[Car89] Luca Cardelli, et al. *Modula-3 Report (revised).* No. 52, Digital Systems Research Center, Palo Alto (CA), 1989.

[FAQ] *General Python FAQ*: "Why was Python created in the first place?"
`http://www.python.org/doc/faq/general.html` (26.3.2004)

[Gri96] Ralph E. Griswold, Madge T. Griswold. *The Icon Programming Language, Third Edition.* Peer-to-Peer Communications, 1996. [Myös `http://www.cs.arizona.edu/icon/lb3.htm`, 26.3.2004]

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.
[Myös `http://www.haskell.org/onlinereport/`, 26.3.2004]

[Ker88] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Inc., 1988.

[Kuc98] Andrew Kuchling. *LJ Interviews Guido van Rossum.* Linux Journal, Issue 55, November 1998. [Myös `http://www.linuxjournal.com/article.php?sid=2959`, 26.3.2004]

[Lis81] Barbara Liskov, et al. *CLU Reference Manual.* Lecture Notes in Computer Science 114, Springer-Verlag, Berlin, 1981.

[McC60] John McCarthy. *Recursive functions symbolic expressions and their computation by machine, Part I.* Communications of the ACM, Volume 3, Issue 4 (April 1960), sivut 184-195. ACM Press, New York, USA.

[Mee84] L.G.L.T. Meertens, S. Pemberton. *Description of B.* CWI Note CS-N8405, CWI, Amsterdam, 1984.

[Mee92] L.G.L.T. Meertens, S. Pemberton, G. van Rossum. *The ABC structure editor: Structure-based editing for the ABC programming environment.* CWI Report CS-R9256, sivu 2, CWI,

Amsterdam, 1992. [Myös
http://www.cwi.nl/ftp/CWIreports/AA/CS-R9256.ps.Z,
26.3.2004]

[PEP1] Barry A. Warsaw, Jeremy Hylton, David Goodger. *Python Enhancement Proposal 1: PEP Purpose and Guidelines.*
http://www.python.org/peps/pep-0001.html (26.3.2004)

[PEP202] Barry Warsaw *Python Enhancement Proposal 202: List Comprehensions.*
http://www.python.org/peps/pep-0202.html (28.3.2004)

[PEP234] Ka-Ping Yee, Guido van Rossum. *Python Enhancement Proposal 234: Iterators.*
http://www.python.org/peps/pep-0234.html (28.3.2004)

[PEP255] Neil Schemenauer, Tim Peters, Magnus Lie Hetland. *Python Enhancement Proposal 255: Simple Generators.*
http://www.python.org/peps/pep-0255.html (28.3.2004)

[PyPy] *PyPy-projektin kotisivu.*
http://codespeak.net/pypy/ (28.3.2004)

[Pyt] *Python-kielen ja CPython-toteutuksen kotisivu.*
http://www.python.org/ (28.3.2004)

[Ros91] Guido van Rossum. *Python distribution posted to alt.sources.* Uutisryhmässä comp.sys.sgi 20.2.1991.
Message-ID: <2986@charon.cwi.nl>
Google Groups -arkisto: http://groups.google.com/groups?selm=2986%40charon.cwi.nl (26.3.2004)

[Ros03a] Guido van Rossum, Fred L. Drake, Jr. *Python Reference Manual, Release 2.3.3.* PythonLabs, 2003.
http://www.python.org/doc/2.3.3/ref/ref.html
(26.3.2004)

[Ros03b] Guido van Rossum, Fred L. Drake, Jr. *Python Library Reference, Release 2.3.3*. PythonLabs, 2003.
`http://www.python.org/doc/2.3.3/lib/lib.html`
(26.3.2004)

[Ros03c] Guido van Rossum, Fred L. Drake, Jr. *Extending and Embedding the Python Interpreter, Release 2.3.3*. PythonLabs, 2003.
`http://www.python.org/doc/2.3.3/ext/ext.html`
(27.3.2004)

[Sco00] Michael L. Scott. *Programming Language Pragmatics (International student edition)*. Morgan Kaufmann, 2000.

[Tan76] Andrew S. Tanenbaum. *A Tutorial on Algol 68*. ACM Computing Surveys, Volume 8, Issue 2 (June 1976), s. 155-190. ACM Press.

[Ven03] Bill Venners. *The Making of Python: A Conversation with Guido van Rossum, Part I.*
`http://www.artima.com/intv/python.html` (26.3.2004)

A Esimerkkiohjelman lähdekoodi

```
1  import sys, types
2
3  CONVERSION_DICT = {
4      ('cm', 'in'): 1/2.54,              # cm -> inch
5      ('in', 'cm'): 2.54,
6      ('kg', 'lbs'): 2.2046,             # kg -> pound
7      ('lbs', 'kg'): 1/2.2046,
8      ('mk', 'euro'): 1/5.94573,
9      ('euro', 'mk'): 5.94573,
10     ('c', 'f'): lambda x: x*1.8 + 32,  # Celcius -> Fahrenheit
11     ('f', 'c'): lambda x: (x-32) * 0.556
12 }
13
14 def convert(value, fromunit):
15     """Convert 'value', which is a numeric value in unit 'fromunit',
16     using CONVERSION_DICT."""
17     try:
18         value + 1.5, float(value)
19     except TypeError:
20         raise TypeError, 'value must be numeric'
21
22     for (dict_from, dict_to), factor in CONVERSION_DICT.iteritems():
23         if dict_from == fromunit:
24             if type(factor)==types.LambdaType:
25                 result = factor(value)
26             else:
27                 result = value * factor
28             print '%s %s = %.2f %s' % (value, fromunit, result, dict_to)
29             break
30     else:
31         print 'Unrecognized unit:', fromunit
32
33 if __name__=='__main__':
34     args = sys.argv[1:]
35     if len(args)!=2:
36         print 'Usage: unitconv <x> <unit>'
37     else:
38         convert(float(args[0]), args[1])
```