Pieni johdatus Python-ohjelmointikieleen

Jonne Itkonen

1. helmikuuta 2007

Sisältö

Luku 1

Johdanto

Python on alankomaalaisen Guido van Rossumin kehittämä olio-ohjelmointikieli, joka on saatavilla lähes jokaiseen käyttöjärjestelmään (koska kielen lähdekoodi on julkinen, saa Pythonin jokaiseen järjestelmään, johon on olemassa ANSI-C kääntäjä). Kieli on kehitetty ABC, Modula-3, C ja Icon kielien pohjalta nopeitten prototyyppien tekoon ja tehokkaammaksi komentojonokieleksi (scripting language). Kieli on tulkkaava[1] ja helposti laajennettavissa joko Python-kielisillä moduleilla tai C/C++-kielisillä laajennuksilla. Nämä laajennokset voivat olla myös dynaamisia, sillä Python tukee useista käyttöjärjestelmistä löytyviä dynaamisesti linkitettäviä kirjastoja. Kielen voi myös sisällyttää oman ohjelman makrokieleksi (esim. Microsoftin Windows-käyttöjärjestelmän yksi Python-toteutus on tehty DLL-kirjastoksi).

Tässä työssä tutustutaan Pythonin syntaksiin ja valmiisiin aliohjelmakirjastoihin. Lisää tietoa kielestä löytyy van Rossumin kirjoittamista monisteista, jotka yleensä toimitetaan Python-toteutusten mukana, sekä World Wide Webistä Pythonin kotisivulta `http://www.python.org/`. Pythonin kotisivulta löytyy tulkin viimeisin versio sekä monisteiden PDF-versiot. Lisätietoja ja viimeiset kuulumiset löytyvät myös uutisryhmästä *comp.lang.python*.

Python on jo saavuttanut kriittiset mitat, jotka estävät kieltä muuttumasta radikaalisti. Kieltä pyritään pitämään vapaana, joten käyttäjä ei joudu maksamaan mitään rojalteja edes myymistään Python-ohjelmista. Tämä lienee osaltaan lisännyt kiinnostusta kieleen niin, että kiinnostusta on alkanut ilmetä Australian puhelinyhtiöstä NASA:n avaruussukkulaohjelmiin asti – Python ei enää ole mikään kokeiluprojekti ;).[2]

Tämä kirjoitus on päivitetty versio luonnontieteen kandidaatin työstä, jonka tein vuonna 1995. Muokkaus on vielä kesken, joten joitain virheitä ja erikoisuuksia saattaa tekstistä löytyä. Kommentteja ja muokkausehdotuksia otetaan ilolla vastaan.

Tämän tekstin ohella suosittelen Pythonin tutoriaalin [4] lukemista. Se on niin hyvä kirjoitus, ettei sen, sekä kielen ja sen kirjastojen referenssimanuaalien lisäksi muuta tarvita kielen opiskeluun. Eli tämän kirjoituksen tarkoitus paljastuu vain ovelaksi juoneksi herättää lukijan kiinnostus kieleen, jonka jälkeen tuon tutoriaalin lukeminen kietoo hänet ikuisesti kiinni tähän mainioon ohjelmointikieleen.

Vastaus: Kielen nimi tulee BBC:n komediasarjasta *Monty Python's Flying Circus*, eikä sillä ole mitään tekemistä niljakkaiden käärmeiden kanssa ;-)

[1] Itseasiassa Python-tulkki kääntää lähdekoodin helpommin ajettavaan muotoon, jota sitten käytetään niin kauan, kunnes lähdekoodissa tapahtuu muutoksia, jolloin se käännetään uudelleen. Python-koodi vaatii aina Python-tulkin, jotta sitä voitaisiin ajaa. Menetelmä on tuttu mm Unixin eri kuorista (ksh, bash, tcsh, ...).

[2] Tämä kommentti on vuodelta 1995. Nykyään tilanne on vieläkin valoisampi, kulkeehan Python jo monella mukana kännykässäkin!

Luku 2

Tulkkiympäristö ja sen käyttö

Python-kielellä kirjoitetut ohjelmat käännetään tavukoodiksi, jota ajetaan virtuaalikoneessa. Tämä on jo nykyään tuttua niin monesta muustakin ohjelmointikielestä, mutta hieman vieraampaa on mahdollisuus käyttää Python-tulkkia vuorovaikutteisesti, kuten vanhoja kunnon Basic-tulkkejakin. Tulkissa voidaan määritellä uusia funktioita ja luokkia ja käyttää valmiita moduleja; tehtyjä funktioita ja luokkia ei voi tallentaa, joten parempi tapa on käyttää tekstieditoria ohjelman tekemiseen ja tulkkia vain ohjelmien testaamiseen ja ajamiseen.

Python käynnistyy joko ikonista tai komentoriviltä komennolla *python*. Python-ohjelmia voidaan ajaa kirjoittamalla python-komennon jälkeen ajettavan tiedoston nimi tai, jos Python-kielinen tiedosto on tehty ajettavaksi, kirjoittamalla pelkästään ko. tiedoston nimen.

Alla on esimerkki Python-tulkin käytöstä laskimena. Huomaa kokonaislukujen ja liukulukujen erot. #-merkki on Pythonissa kommentin aloitusmerkki. Kommentti jatkuu #-merkistä rivin loppuun.

Ensimmäinen esimerkki on tämän kirjoituksen alkuperäisestä versiosta, joka on vuonna 1995 kirjoittajan kirjoittama luonnontieteen kandidaatin tutkielma. Myöhemmissä esimerkeissä käytetään Python toteutusta, jonka versio on vähintään 2.4.

```
tarzan:/users/ji>python
Python 1.2 (Feb 13 1995)
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> 2+2
4
>>> 3*(2+3/4)     # Lasketaan jakolasku kokonaisluvuilla.
6
>>> 3.*(2.+3./4.) # Ja sama liukuluvuilla.
8.25
>>> a=175         # Voidaan myös käyttää 'muuttujia'.
>>> a*3
525
>>> a/2           # Huomaa kokonaisjako.
87
>>> a/2.
87.5
```

Moduleissa esiteltyjä funktioita voidaan käyttää joko esittelemällä koko moduli tai vain tarvittavat modulin funktiot (tai vakiot).

```
>>> import math
>>> math.sin(math.pi/2)
1.0
>>> from math import sin,pi
>>> sin(pi/4)
0.707106781187
```

Tulkkiympäristön yksi mukavimmista ominaisuuksista on *help()*-funktio.

```
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> help()

Welcome to Python 2.5!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://www.python.org/doc/tut/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Help-kehoitteesta pääsee pois antamalla syötteen *quit* tai tyhjän syötteen. Syötteellä *modules* saa apua Python-ympäristön moduleista, ja *keywords* kielen avainsanoista. Myös olioille ja arvoille voi kutsua *help()*-funktiota.

```
>>> help(1)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 .
 .  ...
 .
 |  __new__ = <built-in method __new__ of type object at 0x71d9c0>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T

>>>
```

Merkkijonoille tämä ei tietenkään toimi, sillä mistä tiedettäisiin, pyydetäänkö avustusta merkkijono-oliosta vai sen sisällöstä.

```
>>> help(’’)

>>> help(’string’)
Help on module string:

NAME
    string - A collection of string operations (most are no longer used).
...
    whitespace = ’\t\n\x0b\x0c\r ’


>>> help(type(’’))
Help on class str in module __builtin__:

class str(basestring)
 |  str(object) -> string
 .
 .  ...
 .
 |  __new__ = <built-in method __new__ of type object at 0x721820>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T

>>>
```

Luku 3

Muuttujat

Pythonissa muuttujat ovat nimiä, jotka sidotaan johonkin olioon (aliasing). Tämä on aluksi ehkä vaikea asia tajuta, eikä sen hyviä puolia heti ymmärrä (katso kappaletta 9).

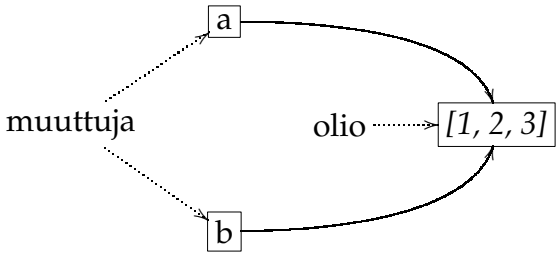
```
>>> a=5
>>> a
5
>>> b=a
>>> b
5
>>> b=b+1
>>> b
6
>>> a
5
```

Viittaukset muuntumattomiin olioihin käyttäytyvät tutulla tavalla, mutta jos viittauksen kohteena on muuntuva olio, on käytös hieman erilaista:

```
>>> a=[1,2,3]      # a viittaa listaolioon [1,2,3]
>>> a
[1, 2, 3]
>>> b=a            # b viittaa a:n viittaamaan olioon
>>> b
[1, 2, 3]
>>> b[1]=4         # muutettaessa b:n alkion arvoa...
>>> b
[1, 4, 3]
>>> a              # muuttuu myös a:n alkion arvo
[1, 4, 3]
>>>
```

Kuva 3 näyttää tilanteesta graafisen esityksen. Sijoituksen *b=a* jälkeen molemmat muuttujat *a* ja *b* osoittavat samaan listaolioon. Täten, kun *b*:hen tehdään muutoksia, muuttuu myös *a*; todellisuudessahan muutoksia tehdään vain listaolioon *[1,2,3]* viitteiden *a* ja *b* kautta[1].

[1]Olisikin parempi puhua viitteistä muuttujien sijaan, mutta koska tässä tekstissä käsitellään vain Pythonia, jossa kaikki muuttujat ovat viitteitä, on *muuttuja*-sanan käyttö oikeutettua.



Kuva 3.1: Muuttujat Pythonissa ovat viittauksia olioihin.

Luku 4

Perustyypit

Pythonissa on kahdenlaisia tietotyyppejä: perustyyppejä ja oliotyyppejä. Perustyypit edustavat arvoja tai arvo-olioita, oliotyypit varsinaisia olioita. Ero arvoille ja olioille on helpointa ymmärtää olion tilan ja identiteetin käsitteiden kautta: arvon tila ja identiteetti ovat sama asia, kun taas oliolle ne ovat kaksi eri asiaa. Esimerkiksi kokonaisluku yksi voidaan identifioida itsellään, sillä ei ole mieltä esim. laskea, kuinka monta kokonaislukua yksi on olemassa. Olioiden määrää taas on mielekästä laskea. Esimerkiksi listaolioita *[1, 2, 3]* voi samassa ohjelmassa olla useita erillisiä. Niitä ei erota niiden tilasta, mutta kylläkin niiden identiteetistä. Arvo-oliot taas ovat olioita, jotka käyttäytyvät kuten arvot. Tyypillinen esimerkki arvo-oliosta on merkkijono. Jos sen sisältöä muuttaa, ei se ole enää sama merkkijono.

4.1 Luvut

Luvut jakautuvat Pythonissa neljään tyyppiin, jotka ovat kokonaisluku, pitkä kokonaisluku, liukuluku ja kompleksiluku. Kokonaisluvut on toteutettu C-kielen long tyypillä, joten niiden tarkkuus on vähintään 32-bittiä. Pitkillä kokonaisluvuilla ei ole tarkkuuden ylärajaa. Liukuluvut on toteutettu C-kielen double-tyypillä, niiden tarkkuus on siten koneriippuvainen (kääntäjäriipppuvainen?).

Tavallinen kokonaisluku on pelkkä numerosarja. Siitä saadaan pitkä kokonaisluku lisäämällä perään merkki 'L' tai 'l', joista ensimmäinen on luettavuussyistä suositeltavampi (vertaa 1l ja 1L).

Luvuille on voimassa tavalliset aritmeettiset operaatiot $+$, $-$, $*$, $/$ ja $\%$ (jakojäännös). Python tukee myös seka-aritmetiikkaa muuntaen laskettaessa aina pienemmän tyypin suuremmaksi, eli esimerkiksi kokonaisluvut muuttuvat pitkiksi kokonaisluvuiksi, jotka muuttuvat liukuluvuiksi. Tyypinmuunnoksia voi tehdä myös funktioilla *int()*, *long()* ja *float()*.

Lisäksi käytössä on funktiot *abs(x)*, *divmod(x,y)* ja *pow(x,y)*, jotka antavat luvun x itseisarvon, parin (x/y, x%y) ja x:n korotettuna potenssiin y.

```
>>> 2+3
5
>>> 2.3*2
4.6
>>> 2**4
16
>>> 2**128
340282366920938463463374607431768211456L
>>> 7/2
```

```
3
>>> 7/2.0
3.5
>>> 7%2
1
>>> divmod(7,2)
(3,1)
>>> pow(2,4)
16
```

Kokonaisluvuille on myös määritelty binääriset operaatiot | (tai), ∧ (poissulkeva tai), & (ja), << (siirto vasemmalle), >> (siirto oikealle), ~ (bittien kääntäminen).

Moduli *math* sisältää yleisimmät matemaattiset funktiot.

Kompleksilukuja käytetään seuraavasti

```
>>> complex(0,1)     # kaksi eri tapaa merkitä kompleksilukua
1j
>>> 0+1j             # tässä toinen tavoista
1j
>>> (0+1j)              # sulkuja voi käyttää selkeyden vuoksi
1j
>>> 1j*complex(0,1)
(-1+0j)
>>> 1j*1j
(-1+0j)
>>> a=complex(1,2)
>>> a.real           # reaaliosan haku
1.0
>>> a.imag           # imaginaariosan haku
2.0
```

4.2 Boolean totuusarvot

Nykyisistä Pythonin versioista löytyy Boolean totuusarvot tosi ja epätosi. Loogiset vertailut palauttavat tuloksensa Booleaneina. Vanhoissa Pythonin versioissa, joista Booleaneja ei löydy, tosi-arvoa vastaa kokonaisluku yksi ja epätosi-arvoa kokonaisluku nolla. Yhteensopivuuden vuoksi näitä voi vieläkin käyttää Boolean-arvojen tilalla, mutta uusien ohjelmien tulisi käyttää vain Booleaneja.

```
>>> 1 > 0
True
>>> 1 < 0
False
>>> 1 == True
True
>>> 0 == True
False
>>> 0 == False
True
```

```
>>> type(True)
<type 'bool'>
```

4.3 Sarjatyypit

Pythonissa on kahdenlaisia sisäänrakennetuja sarjatyyppejä: muuntuntumattomia ja muuntuvia (immutable and mutable sequences). Muuntumattomia ovat merkkijonot ja sarjat, muuntuvia listat. Näiden erona on se, ettei muuntumattomien sarjojen kokoa tai alkioiden arvoa voida muuttaa. Sarjan alkioiden ei kuitenkaan tarvitse olla samaa tyyppiä, esimerkiksi listaan voi laittaa sekaisin lukuja ja merkkijonoja, sekä tietysti toisia listoja (tämä pätee myös monikoille, muttei merkkijonoille – millainen olisi listan sisältävä merkkijono?).

Sarjojen lisäksi Pythonissa on yksi sisäänrakennettu karttatyyppi (mapping type) sanasto (dictionary). Sanastot ovat muuntuvia, ja niiden erona listoihin on se, että sanaston indekseinä voivat toimia muuntumattomat tietotyypit, eli numerot, merkkijonot, monikot ja jotkut luokkien esiintymät (siis oliot). Indekseinä ei voi käyttää toisia sanastoja tai listoja.

Jos kartta- tai sarjatyyppisen olion indeksillä ei ole arvoa (tai indeksiä ei löydy), antaa Python virheilmoituksen.

4.4 Merkkijonot

Merkkijonot ovat Pythonissa sisäänrakennettu muuntumaton tyyppi – jokainen merkkijono on olio (tarkemmin arvo-olio) merkkijonotyypin ilmentymä. Alla on esitelty merkkijonojen ominaisuudet, joista suurin osa on yleisesti muuntumattomien sarjojen ominaisuuksia.

Varsinaista merkkitietotyyppiä Pythonissa ei ole, vaan merkit esitetään yksialkioisina merkkijonoina tai tarvittaessa kokonaislukuina, jotka vastaavat merkin koodia käytetyssä merkkikoodistossaa.

Merkkijono annetaan heittomerkkien tai lainausmerkkien ympäröimänä. Heittomerkkien sisällä olevia lainausmerkkejä ei huomioida. Tämä pätee myös toisinpäin.

```
>>> a='Kissa istuu puussa!'
>>> a
'Kissa istuu puussa!'
>>> a='"Miaaauuuuu", naukaisi kissa.'
>>> a
'"Miaaauuuuu", naukaisi kissa.'
```

Useampirivinen merkkijono ympäröidään kolmella perättäisellä lainaus- tai heittomerkillä. Merkintä `\n` on Java- ja C-kielestä tuttu, ja tarkoittaa rivinvaihtoa.

```
>>> a="""Usean
... rivin
... merkkijono."""
>>> a
'Usean\nrivin\nmerkkijono.'
>>> print a
Usean
rivin
```

```
merkkijono.
>>>
```

Merkkijonon pituus saadaan sisäänrakennetulla funktiolla *len(a)*:

```
>>> len(a)
19
```

Merkkijonon alkioihin (kirjaimiin) voidaan viitata antamalla merkkijonon muuttujan perässä alkion indeksi hakasulkeissa - ensimmäisen alkion indeksi on nolla, kuten C:ssä.

```
>>> a[3]
's'
```

Merkkijonon kirjaimia ei voi muuttaa indeksien avulla, vaan on luotava uusi merkkijono yhdistämällä merkkijonon alku, uusi merkki ja loppu +-operaatiolla. Tämä johtuu siitä, että merkkijono on muuntumaton tyyppi. Tosin on äärimmäisen helppoa luoda oma merkkijono-olio, joka sallii tämän operaation.

Merkkijonoja voidaan myös paloitella antamalla hakasulkeissa kaksoispisteellä eroteltuna ensimmäisen merkin indeksi ja viimeistä seuraavan merkin indeksi[1], eli *a[i:j]* palauttaa merkkijonosta *a* merkit indekseillä *x*, missä $i \leq x < j$.

```
>>> a[6:11]
'istuu'
a[i:] palauttaa merkit i:nnestä merkistä eteenpäin merkkijonon loppuun asti.
>>> a[3:]
'sa istuu puussa!'
a[:j] palauttaa merkkijosta a j ensimmäistä merkkiä.
>>> a[:3]
'Kis'
Täten a[:i]+a[i:] palauttaa alkuperäisen merkkijonon.
>>> a[:3]+a[3:]
'Kissa istuu puussa!'
```

Negatiiviset indeksit antavat merkin paikan laskettuna merkkijonon lopusta, eli *a[i:j]*, missä *i<0*, tarkoittaa samaa kuin *a[len(a)+i:len(a)+j]*, silti -0 on sama kuin 0. Alla olevat esimerkit selvittävät negatiivisten indeksien käyttöä. #-merkki aloittaa Pythonissa kommentin, joka jatkuu rivin loppuun (vrt. C++:n ja Javan //).

```
>>> a[-1]  # palauttaa merkkijonon viimeisen kirjaimen
'!'
>>> a[-5:] # palauttaa viimeiset viisi kirjainta
'ussa!'
>>> a[:-5] # palauttaa 5. viimeiseen merkkiin asti
'Kissa istuu pu'
>>> a[3:-5]
'sa istuu pu'
>>> a[3:5]
'sa'
```

[1]Tämä tapa merkkijonojen *viipaloimiseen* (slicing) on sama kuin Icon-kielessä, ehkä jopa kotoisin sieltä.

Indeksien on kuitenkin syytä olla järjestyksessä. Ensimmäisen indeksin on osoitettava merkkijonossa paikkaan ennen jälkimmäisen indeksin osoittamaa paikkaa.

```
>>> a[-3:5]
''
```

Merkkijonoja voidaan katenoida +-operaatiolla.

```
>>> 'Kissa'+' '+'istuu'
'Kissa istuu'
```

Merkkijonoja voi kertoa kokonaisluvulla, positiiviset katenoivat merkkijonon itsensä kanssa, negatiivisilla tulos on tyhjä merkkijono.

```
>>> 'Kissa'*3
'KissaKissaKissa'
>>> 3*'Kissa'
'KissaKissaKissa'
```

Funktio *min(l)* palauttaa merkkijonon pienimmän, funktio *max(l)* suurimman alkion.

```
>>> min(a)+max(a)
' u'
```

Lause *m in mjono* palauttaa arvonaan *True*, jos merkki *m* löytyy merkkijonosta *mjono*, muuten palautuu *False*. Vanhat Python-versiot palauttivat tosi-arvon tilalla ykkösen ja epätoden arvon tilalla nollan.

```
>>> 's' in 'Kissa'
1
>>> 'u' in 'Kissa'
0
>>> 'u' not in 'Kissa'  # not in - merkki ei kuulu merkkijonoon
1
>>> 'k' in 'Kissa'
0
```

Python osaa myös käsitelllä Unicode-merkistöä käyttäviä merkkijonoja. Tällöin merkkijonon eteen tulee lisätä tarkenteeksi u-kirjain. Merkit, joita ei lähdekoodin joukkoon voida muuten kirjoittaa, voidaan esittää Unicode-merkkijonon sisällä antamalla niiden Unicode-merkkikoodi merkintänä `\uXXXX`, missä XXXX korvataan merkin koodilla. Esimerkiksi komennolla **print** `u'\u20ac'` tulostuu euromerkki '€'.

Merkkijonojen käsittelyyn on tarjolla myös säännölliset lausekkeet (regular expressions), mutta niitä ei käsitellä tässä monisteessa. Muista merkkijonojen ominaisuuksista mainittakoon vielä metodi *split()* ja formaattikoodien käyttö merkkijonoissa. Formaattikoodeista on kerrottu *print*-komennon yhteydessä (kts. 5.1).

Metodilla *split()* merkkijono voidaan paloitella osiin halutun erotinmerkkijonon esiintymiskohdista. Metodi palauttaa listan merkkijonoja, jotka se kohdemerkkijonosta löysi.

```
>>> '1,2,3,4'.split(',')  # erotetaan pilkulla erotellut numerot
['1', '2', '3', '4']
>>> '1,2,3,4'.split('X')  # jos erotinta ei löydy, palautuu koko merkkijono
['1,2,3,4']
>>> 'Kissa istuu puussa.'.split()      # oletuksena erotin on tyhjät merkit
['Kissa', 'istuu', 'puussa.']
>>> 'Kissa istuu puussa.'.split(' .')  # koko merkkijono toimii erottimena
['Kissa istuu puussa.']
>>> '1, 2, 3, 4'.split(', ')    # välilyönti mukana erottimessa
['1', '2', '3', '4']
>>> '1, 2, 3, 4'.split(',')     # ei välilyöntiä erottimessa
['1', ' 2', ' 3', ' 4']        # huomaa välilyönnit
```

Merkkijonoja sisältävän listan alkiot voi kätevästi yhdistää yhdeksi merkkijonoksi metodilla *join()*.

```
>>> a='1,2,3'.split(',')
>>> a
['1', '2', '3']
>>> '|'.join(a)
'1|2|3'
```

4.5 Listat

Listat määritellään luettelemalla listan alkiot hakasulkujen välissä. Listatyyppi on yksi Pythonin muuttuvista sarjatyypeistä (mutable sequences). Listan alkioiden ei tarvitse olla samantyyppisiä olioita. Alla on esimerkein selitetty osa listaolion metodeista ja ominaisuuksista.

```
>>> lista=[1,2,3]
>>> lista
[1, 2, 3]
>>> lista2=[1,2.0,'kissa',a,l,[4,5,6]
>>> lista2
[1, 2.0, 'kissa', 'Kissa istuu puussa!', [1, 2, 3], [4, 5, 6]]
```

Listan yksittäisiä alkioita voidaan lukea tai muuttaa, lisätä tai poistaa. Listan alkioon viitataan antamalla alkion numero hakasulkeissa. Kuten merkkijonoissa, on listassa ensimmäisen alkion indeksinä 0.

```
>>> lista[0]
1
>>> lista[-1]
3
>>> lista[1]=5
>>> lista
[1, 5, 3]
```

Listaan lista voi lisätä alkion metodilla *append(a)*, joka lisää alkion *a* listan loppuun, tai metodilla *insert(i,a)*, joka lisää alkion *a* indeksin *i* kohdalle. Jos indeksi $i \leq 0$, lisätään alkio listan alkuun, jos taas $i \geq listan\ pituus$, lisätään alkio listan loppuun.

```
>>> lista.append(4)
>>> lista
[1, 5, 3, 4]
>>> lista.insert(1,2)
>>> lista
[1, 2, 5, 3, 4]
>>> lista.insert(175,'loppu')
>>> lista.insert(-175,'alku')
>>> lista
['alku', 1, 2, 5, 3, 4, 'loppu']
```

Alkioita voidaan poistaa listasta komennolla *del* antamalla sille parametriksi poistettava alkio.

```
>>> del lista[0]
>>> del lista[-1]
>>> lista
[1, 2, 5, 3, 4]
```

Listan *lista* pituuden saa selville Pythonin sisäänrakennetulla funktiolla *len(lista)*.

```
>>> len(lista)
5
```

Kaikki muuntumattomille sarjatyypeille määritellyt operaatiot toimivat listoille.

```
>>> lista=[1,2,3]*3
>>> lista
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> lista [3:6]=[4,5,6]
>>> lista
[1, 2, 3, 4, 5, 6, 1, 2, 3]
>>> del lista[6:9]
[1, 2, 3, 4, 5, 6]
>>> lista=lista+[7,8,9]
>>> lista
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> 5 in lista
1
```

Näiden lisäksi muuntuvilla sarjoilla on seuraavat operaatiot. Metodi *count(a)* kertoo alkion *a* lukumäärän listassa. Metodi *index(a)* antaa alkion *a* ensimmäisen esiintymän indeksin listassa.

```
>>> lista=[1,2,3]*3
>>> lista
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> lista.count(3)
3
>>> lista.index(3)
2
```

Alkio *a* poistetaan listasta lista metodilla *remove(a)*.

```
>>> lista.insert(3,4)
>>> lista
[1, 2, 3, 4, 1, 2, 3, 1, 2, 3]
>>> lista.remove(4)
>>> lista
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Lista lista voidaan järjestää[2] metodilla *sort()* ja kääntää metodilla *reverse()*.

```
>>> lista.sort()
>>> lista
[1, 1, 1, 2, 2, 2, 3, 3, 3]
>>> lista.reverse()
[3, 3, 3, 2, 2, 2, 1, 1, 1]
```

Muunneltavien sarjatyyppien lisäksi Pythonissa on muuntumattomia sarjatyyppejä. Aikaisemmin esitelty merkkijono on toinen näistä, toinen on monikko (n-pari, tuple).

4.6 Monikko

Monikko on kuten lista, mutta sen koko ja alkioiden arvot eivät voi muuttua. Monikko esitellään luettelemalla alkiot sulkujen välissä. Kuten listassa, monikossakin voi olla useamman kuin yhden tyyppisiä olioita. Kaksialkioisesta monikosta käytetään nimitystä *pari*.

```
>>> t=(1, 'kissa', 2)
>>> t
(1, 'kissa', 2)
>>> t[1:3]
('kissa', 2)
```

Listan voi muuttaa monikoksi funktiolla *tuple()*. Monikon muuttaminen listaksi taas tapahtuu funktiolla *list()*.

Monikkojen hyöty selviää seuraavasta esimerkistä, jossa joukko arvoja pakataan ensin monikoksi, joka sitten puretaan.

```
>>> m = 1, 2, 3
>>> m
(1, 2, 3)
```

[2]Jos lista sisältää itse tehtyjä olioita, kannattaa varmistua siitä, että niille on määritelty järjestys (myös muiden olioiden suhteen). Tästä lisää myöhemmin.

```
>>> a, b, c = m
>>> a
1
>>> b
2
>>> c
3
```

Myös listoja voi purkaa samalla tavalla.

```
>>> a, b, c=[4, 5, 6]
>>> a
4
>>> b
5
>>> c
6
```

4.7 Sanastot

Sanastot ovat ainut Pythoniin sisäänrakennettu kartoitustyyppi (mapping type). Sanasto muodostuu aaltosulkeiden välissä olevista avain–arvo -pareista, jotka on eroteltu toisistaan pilkulla. Parien syöttöjärjestyksellä ei ole väliä; sanastossa ne esiintyvät jossain määräämättömässä järjestyksessä, joka ei ole pysyvä. Avain ja arvo saavat olla mitä tahansa järjestyvää, ei muuntuvaa tyyppiä.

```
>>> s={'vesku':2742, 'vesa':2722, 'jonne':2742}
>>> s
{'vesa': 2722, 'jonne': 2742, 'vesku': 2742}
```

Alkioihin viitataan antamalla avain hakasulkeissa.

```
>>> s['vesku']
2742
>>> s['vesku']=2739
>>> s
{'vesa': 2722, 'vesku': 2739, 'jonne': 2742}
```

Sanastoon lisätään uusi avain, kun tehdään sijoitus olemattomalle avaimelle.

```
>>> s['olli']=2742
>>> s
{'vesa': 2722, 'vesku': 2739, 'jonne': 2742, 'olli': 2742}
```

Metodi s.keys() palauttaa listan sanaston s avaimista, s.values() vastaavasti listan sanaston s arvoista. Metodi s.items() palauttaa listan pareista, joiden ensimmäisenä alkiona on avain ja toisena avainta vastaava arvo.

```
>>> s.keys()
['vesa', 'vesku', 'jonne', 'olli']
>>> s.values()
[2722, 2739, 2742, 2742]
>>> s.items()
[('vesa', 2722), ('vesku', 2739), ('jonne', 2742), ('olli', 2742)]
```

Avaimen *k* esiintymistä sanastossa *s* voidaan testata metodilla *has_key(k)*, joka palauttaa ykkösen, jos avain löytyy, ja nollan, jos avainta ei löydy.

```
>>> s.has_key('jonne')
1
>>> s.has_key('quido')
0
```

Avain ja sen arvo voidaan poistaa sanastosta del-komennolla.

```
>> del s['jonne']
>> s
'vesa': 2722, 'vesku': 2739, 'olli': 2742}
```

Jos avainta ei löydy, antaa Python virheilmoituksen.

4.8 Muut sisäänrakennetut tyypit

Pythonin muut tietotyypit on lueteltu alla, niistä on esimerkkejä myöhemmissä kappaleissa.

Modulit Modulit on käsitelty myöhemmin.

Luokat ja luokkainstanssit (oliot) Luokat ja oliot on käsitelty myöhemmin.

Funktiot Funktiot on käsitelty myöhemmin.

Metodit Metodit ovat luokan sisäisiä funktioita, (funktio)attribuutteja. Katso lisää funktioiden ja metodejen eroista luokkia käsittelevästä kappaleesta, sekä "Python Refernce Manual"- ja "Python Library Reference"-monisteista ([3] ja [2]).

Koodioliot Koodioliot ovat käännettyjä Python-ohjelmia. Niitä voidaan luoda *compile()*-funktiolla ja ajaa *exec*-komennolla tai *eval()*-funktiolla.

Tyyppioliot Tyyppiolio määrittelee olion tyypin. Olion tyyppi saadaan selville *type()*-funktiolla. Sisäänrakennettujen tyyppien oliot löytyvät modulista *types*.

Nollaolio Funktio, joka ei näennäisesti palauta mitään arvoa, palauttaa nollaolion. Pythonissa on määritelty yksi nollaolio, *None*.

Tiedosto-oliot Tiedosto-oliot on toteutettu C-kielen stdio-kirjastoa käyttäen. Tiedosto-oliot ja niiden metodit on esitelty sisäänrakennetun funktion *open()* yhteydessä.

Luku 5

Komennot

Pythonissa ei ole kovinkaan monta kieleen sisäänrakennettua komentoa. Sen sijaan funktioita ja moduuleita on riittämiin. Tällainen tapa rakentaa pieni kieli on tuttua C-kielestä. Hyvää siinä on kielen laajennettavuus ja päivitettävyys, huonoa mahdollisuus useiden erilaisten toistensa kanssa päällekkäisten kirjastojen ja funktioiden olemassaoloon.

Seuraavaksi on lueteltu Pythonin komennot ja selitetty niiden toiminta. Myös komennon syntaksi on esitelty.

5.1 print

Muoto:

⟨*print_stmt*⟩ ::= 'print' ([⟨*expression*⟩ (',' ⟨*expression*⟩)* [',']]
| '>>' ⟨*expression*⟩ [(',' ⟨*expression*⟩)+ [',']])

Komennolla *print* tulostetaan olio tai useampia olioita, jotka evaluoituvat lauseesta tila. Huomaa, että pilkulla eroteltujen tulostettavien olioiden väliin tulee välilyönti.

```
>>> print 2
2
>>> lista=[1,2,3]
>>> print 2,(1,2,3),'kissa',lista
2 (1,2,3) kissa [1,2,3]
```

Komento *print* kutsuu olion *__str__()*-metodia, ja tulostaa tämän palauttaman merkkijonon. Jos print-komennon ei haluta tulostavan rivinvaihtoa, tulee viimeisen parametrin jälkeen kirjoittaa pilkku. Tästä on esimerkki for-silmukan yhteydessä.

Jos tulostusta halutaan muokata, voidaan käyttää C-kielestä tuttuja formaattikoodeja. Tällöin print-komennon ensimmäisenä parametrina on formaattimerkkijono, %-merkki ja tämän jälkeen tulostettavat arvot monikossa pilkuilla eroteltuina.

```
>>> print 'Luku: %3d Sana: %s' % (42, 'kissa')
Luku:  42 Sana: kissa
```

C-kielen formaatteja *%n* ja *%p* ei ole, muut toimivat, myös *-merkki formaatissa. Jos *%s*-koodia vastaava muuttuja ei ole merkkijono, koetetaan se muuttaa siksi *str()*-funktiolla. Jos formaattia ja muuttujaa ei saada täsmäämään, aiheutetaan poikkeutus, eikä *core dumpia*.

Formaattimerkkijonossa voi myös olla viittauksia muuttujiin.

```
>>> puh={'Jonne':2742,'Vesku':2739}
>>> print 'Jonnen puhelinnumero on %(Jonne)d ja Veskun %(Vesku)d.' & puh
Jonnen puhelinnumero on 2742 ja Veskun 2739.
```

Tämä on hyödyllinen ominaisuus sisäänrakennetun funktion *vars()*-kanssa, joka palauttaa sanaston kaikista lokaaleista muuttujista.

Nykyisissä Python-versioissa voidaan tiedostoon (tietovirtaan) tulostaa *print*-komennolla.

```
>>> tiedosto = open('/tmp/koe','wt') # avataan tiedosto kirjoittamista varten
>>> print >> tiedosto, 'kissa istuu'
>>> tiedosto.close()
>>> tiedosto2 = open('/tmp/koe','rt') # ... nyt lukemista varten
>>> data = tiedosto2.read()           # luetaan kerralla koko tiedosto...
>>> print data                        # ... ja tulostetaan se
kissa istuu

>>> tiedosto2.close()
>>> data
'kissa istuu\n'
```

Yllä kannattaa huomata, että kirjoittava *print*-komento on lisännyt rivinvaihdon tiedostoon kirjoittamansa merkkijonon perään, ja tulostava *print*-komento lisää vielä toisen tulostukseen. Jos *print*-komennon ei haluta tätä rivinvaihtoa lisäävän, tulee komentolauseen loppuun lisätä pilkku.

5.2 del

Muoto:

⟨*del_stmt*⟩ ::= 'del' ⟨*target_list*⟩

Tuhoaa listan kohteet alkiot rekursiivisesti vasemmalta oikealle. Jos alkio on viittaus, poistetaan tämä viittaus. Jos alkio on viite attribuuttiin, indeksoitu tai viipale, lähetetään del-viesti kyseessä olevalle oliolle.

```
>>> a=5
>>> lista=[1,2,3]
>>> rlista=lista
>>> dir()
[..., 'a', 'lista', 'rlista', ...]
>>> del a
>>> del lista
>>> dir()
[..., 'rlista', ...]
>>> rlista
[1, 2, 3]
>>>
```

Huomaa, ettei listaolio *[1, 2, 3]* katoa, vaikka sen luomiseen käytetty muuttuja lista tuhotaankin. Vasta viimeisen viittauksen tuhoaminen (mahdollisesti) tuhoaa olion[1].

[1]Python-ympäristö tuhoaa olioita vain roskienkeruun yhteydessä. Oliota ei voi varsinaisesti tuhota millään komennolla (*del*-komento tuhoaa nimen sidonnan olioon, itse olio jää olemaan).

5.3 import

Muoto:

⟨*import_stmt*⟩ ::= 'import' ⟨*module*⟩ ['as' ⟨*name*⟩] (',' ⟨*module*⟩ ['as' ⟨*name*⟩])*
| 'from' ⟨*module*⟩ 'import' ⟨*identifier*⟩ ['as' ⟨*name*⟩]
(',' ⟨*identifier*⟩ ['as' ⟨*name*⟩])*
| 'from' ⟨*module*⟩ 'import' '(' ⟨*identifier*⟩ ['as' ⟨*name*⟩]
(',' ⟨*identifier*⟩ ['as' ⟨*name*⟩])* [','] ')'
| 'from' ⟨*module*⟩ 'import' '*'

⟨*module*⟩ ::= (⟨*identifier*⟩ '.')* ⟨*identifier*⟩

Ensimmäinen muoto etsii ja alustaa annetun tunnuksen ilmoittaman modulin ja määrittelee sen sisältämät piirteet (eli esim. funktiot, luokat ja muuttujat) lokaalissa näkyvyysalueessa. Kutsun jälkeen modulin sisältöön viitataan muodossa *tunnus.piirrenimi*.

```
>>> import math
>>> math.sin(math.pi/2)
1.0
```

Toinen muoto määrittelee modulin import-osassa ilmoitetut piirteet lokaalissa näkyvyysalueessa. Piirteisiin viitataan nimellään, tai *as*-tarkentimella annetuilla aliaksilla.

```
>>> from math import pi, sin as sini
>>> sin(pi/2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'sin' is not defined
>>> sini(pi/2)
1.0
```

Kolmas muoto määrittelee kaikki modulissa tunnus määrittellyt piirteet lokaalissa näkyvyysalueessa, paitsi ne piirteet, jotka alkavat alaviivalla '_'.

```
>>> from math import *
>>> cos(pi)
-1.0
```

Tämä viimeinen tapa on huono tapa, sillä se kuormittaa ohjelman nimiavaruuden kaikilla moduulin sisältämien piirteiden nimillä. Käytännössä tällöin moduloinnista ei ole mitään hyötyä. Tuotujen nimien poistaminenkin on työlästä.

Jos *import*-komennolla tuotu moduli muuttuu, saadaan muutokset huomioitua kutsumalla funktiota *reload(moduli)*, jolle annetaan parametrina uudelleenladattavan modulin nimi. Tämä ei toimi, jos modulin piirteitä on yksittäisesti tuotu oman ohjelman osaksi. Tästäkin syystä yllämainituista tavoista ensimmäinen on varsinkin vuorovaikutteista tulkkia käytettäessä se suositeltavin vaihtoehto.

Jos modulia ei löydy, aiheutetaan *ImportError*-poikkeutus. Jos tunnuksen määrittelyssä on syntaksivirhe, aiheutetaan *SyntaxError*-poikkeutus.

5.4 raise

Muoto:

⟨*raise_stmt*⟩ ::= 'raise' [⟨*expression*⟩ [',' ⟨*expression*⟩ [',' ⟨*expression*⟩]]]

Komento *raise* aiheuttaa poikkeutuksen (exception), joka evaluoituu ensimmäisen parametrin mukaan merkkijonoksi, luokaksi tai instanssiksi. Jos ensimmäinen parametri on luokka, on toisen parametrin oltava saman luokan (tai sen perillisen) instanssi. Jos ensimmäinen parametri on luokan instanssi, täytyy toisen parametrin olla *None*.

Jos ensimmäinen parametri on merkkijono tai luokka, aiheuttaa *raise* poikkeutuksen, jonka tunnus ensimmäinen parametri on, parametrinaan toinen parametri (tai *None*). Jos ensimmäinen parametri on instanssi, aiheuttaa *raise* poikkeutuksen, jonka tunnuksena on ensimmäisen parametrin luokka ja parametrina ensimmäisenä parametrina annettu instanssi.

Kolmas parametri on traceback olio. Tätä vaihtoehtoa ei käsitellä tässä kirjoituksessa.

```
>>> raise 'Hätätila', 'Sikanauta lopussa!'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
Hätätila: Sikanauta lopussa!
>>>
```

5.5 pass

Muoto:

⟨*pass_stmt*⟩ ::= 'pass'

Komento *pass* ei tee, eikä aiheuta mitään; siitä on hyötyä, kun halutaan määritellä rakenteita, jotka eivät vielä tee mitään. Myöhemmin ohjelmassa voidaan sitten lisäillä esimerkiksi allaolevassa tapauksessa luokalle Jasen() uusia tieto- ja metodiattribuutteja.

```
>>> class Jasen():
...     pass
...
>>> jasen=Jasen()
>>> dir(j)
[]
>>> jasen.nimi='Mrs P.J.Smegma'
>>> jasen.harrastus='Trondheimiläiset vasaratanssit'
>>> dir(jasen)
['nimi', 'harrastus']
>>>
```

Sisäänrakennettu funktio *dir(a)* palauttaa argumenttinsa *a* *__dict__*-attribuutin, eli argumenttinsa *a* sisällä määritellyt attribuutit. Argumentti a voi olla moduli, luokka tai luokan instanssi (olio).

Hieman selkeämpi käyttökohde *pass*-komennolle on luoda sen avulla abstrakti metodi olion luokalle. Tämä on kuitenkin huono tapa; parempi olisi tehdä abstraktista metodista sellainen, että

se aiheuttaa *raise*-komennolla poikkeutuksen. Metodin perivässä aliluokassa annetaan metodille konkreetti toteutus, ja jos sellainen unohtuu, aiheutuu abstraktin metodin kutsusta poikkeutus yliluokan toteutuksen mukaisesti.

```
>>> class Abstrakti:
...         def foo(self):
...                 pass
...         def bar(self):
...                 raise "Metodia ei ole toteutettu aliluokassa!"
...
>>> a=Abstrakti()
>>> a.foo()             # ei tee mitään, mutta onkin abstrakti
>>> a.bar()             # tämä aiheuttaa poikkeutuksen
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in bar
Metodia ei ole toteutettu aliluokassa!
>>> class Konkreetti(Abstrakti):
...         def bar(self):
...                 print 'Hoplaa!'
...
>>> k=Konkreetti()
>>> k.foo()                 # huonosti käy!
>>> k.bar()
Hoplaa!
>>> class Konkreetti2(Abstrakti):
...         pass
...
>>> k2=Konkreetti2()
>>> k2.foo()               # huonosti käy taas...
>>> k2.bar()               # vaan tästä puutteesta tiedotetaan!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in bar
Metodia ei ole toteutettu aliluokassa!
>>>
```

5.6 return

Muoto:

⟨*return_stmt*⟩ ::= 'return' [⟨*expression_list*⟩]

Komennolla *return* palautetaan funktiosta yksi tai useampia arvoja. Jos *return*-komennolle ei anneta parametreja, se palauttaa nollaolion (*None*). Jos palautettavia arvoja on useita, palautetaan monikko, jonka alkioina ovat palautettavat arvot.

5.7 break ja continue

Muoto:

⟨*break_stmt*⟩ ::= 'break'

⟨*continue_stmt*⟩ ::= 'continue'

Komennot *break* ja *continue* toimivat kuten C-kielen vastaavat komennot. Komennolla *break* poistuu silmukasta, *continue* jatkaa silmukkaa seuraavaan iteraatioon.

5.8 exec

Muoto:

⟨*exec_stmt*⟩ ::= 'exec' ⟨*expression*⟩ ['in' ⟨*expression*⟩ [',' ⟨*expression*⟩]]

Komento *exec* suorittaa Python-kielisen lauseen käyttäen globaalien muuttujien varastona *in*-tarkentimella annettua sanastoa ja lokaalien muuttujien varastona sanastoa, joka on annettu edellisestä sanastosta pilkulla erotettuna. Jos vain ensimmäinen sanasto annetaan, sijoitetaan globaalit ja lokaalit muuttujat sinne, jos kumpaakaan sanastoa ei anneta, suoritetaan lause *exec*-komennon sisältävässä näkyvyysalueessa (scope). Lauseen täytyy evaluoitua joko merkkijonoksi, tiedosto- tai koodiobjektiksi.

```
>>> exec "def Spam(): print 'Spam!'"
>>> Spam
<function Spam at 8bee4>
>>> Spam()
Spam!
```

Lauseen evaluointi voidaan tehdä dynaamisemmin sisäänrakennetulla funktiolla *eval()*.

```
>>> omat_globaalit={}
>>> omat_lokaalit={}
>>> exec 'x=1' in omat_globaalit, omat_lokaalit
>>> omat_globaalit
{'__builtins__': ... }
>>> omat_lokaalit
{'x': 1}
>>> eval('x+1', omat_globaalit, omat_lokaalit)
2
>>> x=3
>>> eval('x+1', omat_globaalit, omat_lokaalit)
2
```

Ylläolevassa esimerkissä kannattaa huomata, että vaikka rivillä 10 on määritelty lokaalin muuttujan *x* arvoksi kolme, ei tämä määritys siirry *eval()*-funktion käyttämään lokaaliin nimiavaruuteen *omat_lokaalit*.

5.9 global

Muoto:

⟨*global_stmt*⟩ ::= 'global' ⟨*identifier*⟩ (',' ⟨*identifier*⟩)*

Määrittelee globaaliksi yhden tai useamman tunnuksen.

5.10 Muita komentoja

Muita Pythonista löytyviä komentoja ovat *yield* ja *with*, mutta näitä ei käsitellä tässä kirjoituksessa tarkemmin. Lyhyesti *yield* mahdollistaa vuorottaisrutiinit (coroutine), ja sitä voidaan käyttää esim. generaattorien toteuttamiseen. Komennolla *with* taas luodaan suorituslohko, joka käyttää erillisiä kontekstia ohjaavia metodeja. Tämä komento on jo käytettävissä Pythonin versiossa 2.5, mutta varsinaisesti se tulee käyttöön vasta versiossa 2.6.

Luku 6

Kontrollirakenteet

Seuraavana on esitelty Pythonin kontrollirakenteet. Vaikka rakenteiden nimet saattavatkin olla tuttuja proseduraalisista kielistä, ovat rakenteiden toiminnat hyvinkin poikkeavia ja yleensä monipuolisempia (vertaa vaikka for-lausetta C/C++:n for-lauseeseen).

6.1 if-lause

Muoto:

⟨*if_stmt*⟩ ::= 'if' ⟨*expression*⟩ ':' ⟨*suite*⟩
('elif' ⟨*expression*⟩ ':' ⟨*suite*⟩)*
['else' ':' ⟨*suite*⟩]

Testaa läpi tiloja *expression* niin kauan, kunnes kyseinen tila on tosi, jolloin suoritetaan tilaa vastaava lohko *suite*. Jos mikään tiloista ei ole tosi, suoritetaan tunnusta *else* seuraava lohko *suite*.

Pythonissa ohjelmalohkoja ei ympäröidä suluilla tai avainsanoilla, vaan ne merkitään sisennyksillä. Tällöin ohjelmakoodista näkee varmasti lohkoon kuuluvan koodin, eikä C-kielen lohkotuksen kaltaisia ongelmia pääse syntymään. Sisennyksissä kumminkin tulee olla tarkkana; mahdolliset ongelmat vähenevät, jos käytetään aina joko sarkainta tai välilyöntejä sisennykseen, ei molempia sekaisin. Pythonin lohko lopetetaan *ulontamalla* rivi, siis poistamalla yksi tai useampi sisennys.

```
>>> x=1
>>> if x==0:
...     print 'Nolla'
... elif x==1:
...     print 'Yksi'
... else:
...     print 'En tiedä!'
...
Yksi
```

6.2 while-silmukka

Muoto:

⟨*while_stmt*⟩ ::= 'while' ⟨*expression*⟩ ':' ⟨*suite*⟩
['else' ':' ⟨*suite*⟩]

Niin kauan kuin tila *expression* on tosi, suoritetaan lohko *suite*. Kun tila on epätosi, suoritetaan mahdollinen avainsanan *else* jälkeinen lohko. Jos silmukasta poistutaan *break*-komennolla, ei *else*-osaa suoriteta.

```
>>> a,b=0,1
>>> while b<1000: # Tulostetaan 1000 pienemmät luvut Fibonaccin sarjasta
...     print b,
...     a,b=b,a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

6.3 for-silmukka

Muoto:

⟨*for_stmt*⟩ ::= 'for' ⟨*target_list*⟩ 'in' ⟨*expression_list*⟩ ':' ⟨*suite*⟩
['else' ':' ⟨*suite*⟩]

Pythonin for-silmukkassa iteroidaan lohkoa *suite* kohdelistan *target_list* alkioilla yli lähdelistan *expression_list*. Lopuksi suoritetaan mahdollinen avainsanan *else* jälkeinen lohko. Komennot *break* ja *continue* toimivat kuten *while*-silmukassa.

```
>>> for i in [1,2,'kissa',(1,2,3)]: print i,
...
1 2 kissa (1, 2, 3)
>>> for i in 'Kissa istuu puussa.':
...    print i,
...
K i s s a   i s t u u   p u u s s a .
```

Ylimääräinen välilyönti kirjainten välissä johtuu Pythonin *print*-komennon ominaisuudesta erotella tulostettavat oliot järkevästi. Tällaisissa operaatioissa on nykyisissä Python-versioissa tapana käyttää listarakentimia (kts. kappale 7 s. 31).

Varoitus! Lähdelistaa *ei saa muokata* silmukan sisällä. Jos sitä tarvitsee muokata, täytyy siitä antaa kopio lähdelistaksi. Tämä johtuu listojen sisäisestä esitystavasta Pythonissa.

```
>>> a=[1,2,-3,4,-5,-3,-5,-5,-3]
>>> for x in a:                  # TÄMÄ ON VÄÄRIN !!!!!!!!
...   if x<0: a.remove(x)
...
>>> a                           # HUOMAA VÄÄRÄ TULOS!
[1, 2, 4, -5, -3]
>>> a=[1,2,-3,4,-5,-3,-5,-5,-3]
>>> for x in a[:]:               # TÄMÄ ON OIKEIN !!!!!!!!
...   if x<0: a.remove(x)
...
>>> a                           # HUOMAA OIKEA TULOS!
[1, 2, 4]
```

for-lauseen apuna käytetään usein sisäänrakennettua funktiota *range()*, joka palauttaa argumenttiensa määrittelemän listan numeroita. Suurille väleille voi käyttää funktota *xrange()*, joka generoi kerralla yhden listan alkion. Sen toiminta vastaa muutoin täysin *range()*:a.

```
>>> range(16)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> range(8,16)
[8, 9, 10, 11, 12, 13, 14, 15]
>>> range(0,16,2)
[0, 2, 4, 6, 8, 10, 12, 14]
>>> for i in range(16):
...     print i,
...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

6.4 try-lause

Muoto:

⟨*try_stmt*⟩ ::= ⟨*try1_stmt*⟩ | ⟨*try2_stmt*⟩

⟨*try1_stmt*⟩ ::= 'try' ':' ⟨*suite*⟩
('except' [⟨*expression*⟩ [',' ⟨*target*⟩]] ':' ⟨*suite*⟩)+
['else' ':' ⟨*suite*⟩]
['finally' ':' ⟨*suite*⟩]

⟨*try2_stmt*⟩ ::= 'try' ':' ⟨*suite*⟩
'finally' ':' ⟨*suite*⟩

try-lause yrittää suoritta lohkon lohko. Jos tapahtuu keskeytys tila, suoritetaan sitä vastaava lohko. Jos kohde on olio, on sen oltava sama olio, joka aiheutti keskeytyksen, eikä pelkästään viittaus siihen. Tilaton *except*-lause tulee olla viimeisenä. Jos *try*-lauseen lohkossa ei tapahdu keskeytystä, suoritetaan *else*-lohko.

Toisessa muodossa, jos *try*-lohkossa tapahtuu keskeytys, tallennetaan se ja suoritetaan *finally*-lohko, jonka suorittamisen jälkeen aiheutetaan tallennettu keskeytys. Jos *try*-lohkosta poistutaan return- tai *break*-komennolla, suoritetaan *finally*-lohko, muulloin sitä ei suoriteta. Komentoa *continue* ei voi käyttää.

```
>>> try:
...   7/0
... except:
...   print 'Virhe'
... else:
...   print 'Hmm?'
...
Virhe
>>> try:
...   7/0
... finally:
...   print 'Lopultakin!'
```

```
...
Lopultakin!
Traceback (innermost last):
  File "<stdin>", line 2, in ?
ZeroDivisionError: integer division or modulo
```

Pelkkä *except*-osio ilman luokka- ja olio-parametria nappaa minkä tahansa poikkeutuksen. Komennolla *else* merkitty osio suoritetaan, jos mitään poikkeutusta ei tule, ja komennolla *finally* merkitty osio aina viimeisenä, tapahtuipa poikkeutus tai ei. Komennon *finally* käyttö yhdessä *except*-lohkojen kanssa on mahdollista Pythonin versiosta 2.5 eteenpäin.

Luku 7

Funktiot

Seuraavassa käsitellään omien funktioiden määrittely ja Pythonin sisäiset funktiot. Huomautettakoon tässä vaiheessa, että Python tukee myös pienissä määrin funktionaalista ohjelmointia sisäisten funktioidensa, kuten *apply()*, *map()* ja *reduce()*, avulla. Yleensä niiden avulla kuitenkin saa tehtyä ei-luettavaa, joskin tehokkaampaa koodia. Niiden tilalle on nykyisissä Python-versioissa tarjolla listoja generoivat muodot.

```
>>> [x+1 for x in (1, 2, 3)]
[2, 3, 4]
>>> '|'.join([str(x) for x in (1, 2, 3)])
'1|2|3'
>>> '|'.join(map(str, (1, 2, 3)))
'1|2|3'
>>> [2*x for x in range(6) if x%2==0]
[0, 4, 8]
```

Funktion määrittely alkaa sanalla def, jonka jälkeen tulee funktion nimi ja sen parametrit suluissa ja kaksoispiste.

```
>>> def hassu(x):
...   print x
...
>>> hassu(3)
3
```

Funktion runko on kirjoitettava sisennettynä – mitään sulkeita tai avainsanoja ei käytetä. Pythonissa funktio voi palauttaa yhden tai useamman arvon, tai ei mitään, kuten yllä. Jos funktio ei näennäisesti palauta mitään, se itseasiassa palauttaa nollaolion *None*. Seuraavassa esimerkissä luku kolme tulostuu funktiosta *hassu()*, ja *print*-komento tulostaa funktiosta palautuvan *None*-olion.

```
>>> print hassu(3)
3
None
```

Funktion parametrit välitetään arvoparametreina, tai tarkemmin: viittauksina olioihin. Tästä seuraa se, ettei parametrin arvon muutos yleensä näy funktion ulkopuolella, paitsi jos parametri on viittaus muuntuvaan olioon, esimerkiksi listaan.

```
>>> def muutu_ja_paluta(lista,luku):
...     lista.append(luku)
...     luku=luku+1
...     return lista,luku
...
>>> lista=[1,2]
>>> a=3
>>> muutu_ja_palauta(lista,a)
([1,2,3], 4)
>>> lista
[1, 2, 3]
>>> a
3
>>> lista2,b=muutu_ja_palauta(lista,a)
>>> lista2
[1, 2, 3, 3]
>>> b
4
```

Funktion parametreille voi antaa oletusarvoja.

```
>>> def koe(x=175):
...     print x
...
>>> koe()
175
>>> koe(43)
42
>>>
```

Funktiolla voi olla vaihteleva määrä parametreja. Vaihteleva parametrimäärä esitellään *:llä parametrin edessä. Parametri on tällöin monikko.

```
>>> def montako(*p):
...     print p       #Tulostetaan parametrit
...     return len(p) #Funktio palauttaa parametriensa määrän
...
>>> montako(2,3,4)
(2, 3, 4)
3
>>> montako(1,'kissa',(1,2), [3,4])
(1,'kissa',(1,2), [3,4])
4
```

Parametrin **p* alkioihin viitataan normaalisti *p[i]*.

Koska funktion parametrit ovat tyypittömiä, antaa Python virheilmoituksen vasta, kun ohjelman ajon aikana funktiolle välitetään argumenttina vääränlainen olio. Tyyppi voidaan varmistaa tutkimalla funktiossa (tai sen ulkopuolella) argumenttien tyyppien soveltuvuus *type()*-funktiolla.

Funktion *type()*-avulla voidaan myös tehdä polymorfisia funktioita selvittämällä ensin argumentin tyyppi, jonka perusteella sitten suoritetaan haluttu ohjelman osa tai aliohjelma. Dynaamisessa ohjelmoinnissa pyritään kuitenkin yleensä toteuttamaan ohjelmat niin, etteivät vääräntyyppiset argumentit aiheuta ongelmia. Voidaan myös käyttää testilähtöistä ohjelmointitapaa varmistamaan, ettei vääräntyyppisiä argumentteja käytetä, tai niistä haittaa aiheudu.

7.1 Sisäänrakennetut funktiot

Seuraavana on lueteltu sisäänrakennetut funktiot; joistakin on annettu esimerkkejä.

abs(x)

Palauttaa x:n itseisarvon. Luku x on Pythonin kokonaisluku, pitkä kokonaisluku, liukuluku tai kompleksiluku.

apply(fkt, arg)

Suorittaa funktion *fkt* parametreinaan monikko *arg*. Parametri *fkt* on kutsuttava olio (callable object, sisäänrakennettu tai käyttäjän määrittelemä funktio, metodi tai luokka-olio) ja arg edellisen parametrilista monikkona. Tämä eroaa muodosta *fkt(arg)* siinä, että jälkimmäisessä tapauksessa *fkt()*:lle välitetään yksi parametri, monikko *arg*, kun taas *apply()*-funktion tapauksessa *fkt*:lle välitetään *len(arg)*-parametria.

chr(i)

Palauttaa yhden merkin, jonka ASCII-koodi on kokonaisluku *i*. Kokonaisluvun *i* tulee olla väliltä [0...255]. Funktion *chr()* käänteisfunktio on *ord()*.

cmp(x, y)

Vertaa olioita *x* ja *y* keskenään, ja palauttaa kokonaisluvun nolla, jos nämä ovat yhtäsuuret. Jos $x < y$, palautetaan negatiivinen kokonaisluku (suositellaan -1:stä), ja jos $x > y$ palautetaan positiivinen kokonaisluku (suositellaan 1:stä). Käyttäjä voi määritellä omille olioilleen metodin *__cmp__()*, jota tämä funktio kutsuu. Katso myös *hash()*-funktio alempana.

coerce(x, y)

Palauttaa parametrit *x* ja *y* kaksialkioisena monikkona (eli parina) ja muunnettuna yleiseksi tyypiksi aritmeettisten operaatioiden tyypinmuunnossääntöjen mukaan.

compile(mjono, tnimi, tyyppi)

Kääntää merkkijonon mjono koodi-olioksi. Koodi-oliot voidaan suorittaa (ajaa) joko *exec*-komennolla tai *eval()*-funktiolla. Parametri *tnimi* kertoo sen tiedoston nimen, josta koodi luetaan, tai *'<string>'*, jos käännettävä koodi on parametrissa *mjono*. Parametrin *tyyppi* arvo on (merkkijono) *'eval'*, jos käännettävä koodi on yksi ilmaisu (expression), ja *'exec'*, jos käännettävä koodi on sarja lauseita.

```
>>> x=1
>>> k=compile('x+1','<string>','eval')
>>> k
<code object ? at 9e884, file "<string>", line 0>
>>> eval(k)
2
>>> x
1
>>> k=compile('x=x+1','<string>','eval')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<string>", line 1
    x=x+1
     ^
SyntaxError: invalid syntax
>>> k=compile('x=x+1\nprint x','<string>','exec')
>>> x
1
>>> k
<code object ? at 8bfe4, file "<string>", line 0>
>>> eval(k)
2
>>> exec k
3
```

delattr(olio, nimi)

Tuhoaa parametrin *olio* attribuutin *nimi*, jos olion määrittely sallii sen. Funktion kutsu *delattr(x,'foobar')* vastaa komentoa *del x.foobar*.

dir()

Ilman parametreja palauttaa listan paikallisen symbolitaulun nimistä. Jos parametrina on moduli, luokka, luokan esiintymä tai yleensä olio, jolla on *__dict__*-attribuutti, palautaa funktio listan olion attribuuttien nimistä. Palautuva lista on järjestetty.

```
>>> import sys
>>> dir()
['sys']
>>> dir(sys)
['argv', 'exit', 'modules', 'path', 'stderr', 'stdin', 'stdout']
```

divmod(a, b)

Palauttaa kokonaisluvuille parin *(a/b, a%b)*, ja liukuluvuille parin *(math.floor(a/b), a%b)*.

eval(lause[, globaali [, lokaali]])

Tulkkaa Python-lauseen lause, ja suorittaa sen käyttäen sanastoja *globaali* ja *lokaali* globaalien ja lokaalien nimien säilytyspaikkana. Jos parametria *lokaali* ei ole, käytetään sen tilalla parametrin *globaali* nimiavaruutta. Jos molemmat jälkimmäiset parametrit puuttuvat, suoritetaan lause ympäristössä, josta *eval()*-funktiota kutsutaan. Palautetaan suoritetun lauseen tulos.

```
>>> x = 1
>>> print eval('x+1')
2
```

Myös käännettyjä koodi-olioita voidaan suorittaa *eval()*-funktiolla. Katso lisää funktion *compile()*-esittelystä. Katso myös exec-lause, *execfile()*- ja *vars()*-funktiot.

execfile(tnimi[, globaali[, lokaali]])

Funktio on vastaava kuin *eval()*-funktiolla, mutta koodi luetaan tiedostosta, jonka nimi on *tnimi*. Palautusarvo on *None*.

filter(fkt, sarja)

Muodostaa uuden sarjan parametrin sarja alkioista. Uuteen sarjaan otetaan mukaan ne vanhan sarjan alkiot, joille funktio *fkt* on tosi. Jos sarja on merkkijono tai monikko, on uusi sarja vastaavasti merkkijono tai monikko, muuten se on lista. Jos funktio *fkt* on *None*, palautetaan ne sarjan alkiot, jotka eivät ole tosia (siis ovat joko epätosia, tyhjiä tai nollia).

```
>>> lista=[0,1,2,3,4]
>>> def parillinen(x):     # Palauttaa toden, jos x on parillinen luku
...   return x%2==0
...
>>> filter(parillinen,lista)
[0, 2, 4]
>>> filter(None,lista)
[1, 2, 3, 4]
```

float(x)

Muuttaa luvun *x* liukuluvuksi Katso funktiot *int()* ja *long()*.

getattr(olio, nimi)

Kutsuu olion *olio* attribuuttia *nimi*, joka annetaan merkkijonona. Tuloksena on kutsutun attribuutin arvo. Kutsu *getattr(x, 'foobar')* vastaa kutsua *x.foobar*.

hasattr(olio, nimi)

Tarkastaa, onko oliolla *olio* attribuuttia *nimi*. Jos sellainen löytyy, palautetaan funktiosta *True*, muuten *False*.

hash(olio)

Palauttaa parametrin *olio* vertailussa käytettävän avainluvun, joka on 32-bittinen kokonaisluku. Avainlukuja käytetään vertailtaessa sanastojen avaimia avaimen etsinnän aikana.

hex(x)

Palauttaa merkkijonona kokonaisluvun *x* heksadesimaaliesityksen.

id(olio)

Palauttaa kokonaisluvun, joka on olion identiteetti. Kahdella yhtäaikaisella oliolla ei voi olla samaa identiteettiä.

input([mjono])

Kysyy käyttäjältä syöttötietoa. Vastaava kuin *eval(raw_input(mjono))*, paitsi että pitkä syöttö voidaan jakaa usealle riville käyttämällä takakenoviivaa \. Parametri *mjono* on vapaaehtoinen kehote.

int(x)

Muuntaa luvun *x* tavalliseksi (lyhyeksi) kokonaisluvuksi. Liukulukujen muunto kokonaisluvuksi tapahtuu käytetyn C-kielen toteutuksen mukaisesti yleensä katkaisuna kohti nollaa. Katso funktiot *long()* ja *float()*.

len(s)

Palauttaa parametrin *s* pituuden (alkioiden määrän). Parametri *s* voi olla merkkijono, lista, monikko tai sanasto.

long(x)

Muuntaa luvun *x* pitkäksi kokonaisluvuksi. Katso funktiot *int()* ja *float()*.

map(fkt, sarja, ...)

Suorittaa funktion *fkt* jokaiselle sarjan *sarja* alkiolle, ja palauttaa listan syntyneistä arvoista. Parametri *sarja* voi olla tyypiltään mitä tahansa sarjatyyppiä. Funktiolla *fkt* tulee olla yhtä monta parametria, kuin on sarjoja *sarja*. Jos joku sarja on toista lyhyempi, korvataan puuttuvat alkiot arvolla *None*. Jos funktio *fkt* on *None*, muodostetaan lista monikoista, joissa jokaisessa on yksi alkio kustakin sarjasta (siis ensin kaikkien sarjojen ensimmäiset alkiot, sitten kaikkien sarjojen toiset alkiot, jne). Palautettava arvo on aina lista, riippumatta parametreina vietävien sarjojen tyypeistä.

```
>>> map(hex,[0,8,16,32,64])
['0x0', '0x8', '0x10', '0x20', '0x40']
```

max(s)

Palauttaa (ei-tyhjän) sarjan *s* suurimman alkion.

min(s)

Palauttaa (ei-tyhjän) sarjan *s* pienimmän alkion.

oct(x)

Palauttaa merkkijonona kokonaisluvun *x* oktaaliesityksen.

open(tnimi[, tila[, pkoko]])

Palauttaa uuden tiedosto-olion. Parametri *tnimi* ilmoittaa tiedoston nimen, *tila* ilmoittaa, aukaistaanko tiedosto lukemista ('r'), kirjoittamista ('w') vai lisäystä ('a') varten. Tiedoston aukaisu kirjoittamista varten tuhoaa tiedoston aiemman sisällön. Tilat 'r+', 'w+' ja 'a+' avaavat tiedoston päivitystä varten, jos Python-tulkin kääntämisessä käytetyn C-kielen stdio-kirjasto tukee tätä ominaisuutta. Kirjain 'b' lisättynä tilan perään aukaisee tiedoston binääritilassa, jos käyttöjärjestelmä erottelee teksti- ja binääritiedostot toisistaan.

Valinnainen parametri *pkoko* ilmoittaa tiedoston puskurin koon; 0 tarkoittaa puskuroimatonta, 1 rivipuskuroitua ja muut positiiviset arvot (noin) ilmoittamansa kokoista puskuria. Negatiiviset puskurikoot tarkoittavat systeemin oletusarvoa, joka on yleensä rivipuskurointi *tty*-laitteille ja täysi puskurointi muille tiedostoille. Puskurikoon määrittelyllä ei ole vaikutusta käyttöjärjestelmissä, joissa ei ole *setvbuf()*-toimintoa.

Funktion *open()* palauttama olio tuntee seuraavat metodit:

close() Sulkee tiedoston.

flush() Tyhjentää tiedoston puskurin, kuten C-kielen stdio-kirjaston funktio fflush().

isatty() Palauttaa ykkösen, jos tiedosto on kytketty tty-laitteeseen, muuten nollan.

read(koko) Lukee enintään *koko* tavua tiedostosta. Jos luetaan EOF tai luettavaa tietoa ei ole, luetaan vähemmän kuin koko tavua. Jos koko unohdetaan, luetaan tavuja tiedoston loppuun (EOF) asti. Funktio palauttaa lukemansa tavut merkkijonona. Jos heti alussa luetaan EOF, palautetaan tyhjä merkkijono.

readline() Lukee kokonaisen rivin tiedostosta. Rivinvaihtomerkki säilyy palautettavassa merkkijonossa, paitsi jos luettu rivi jää kesken. Päinvastoin kuin stdio-kirjaston *fgets()*-funktio, *readline()* säilyttää tiedostosta luetut nollamerkit '\0' merkkijonossa.

readlines() Lukee rivejä *readline()*-metodilla, kunnes saavutetaan EOF. Luetut rivit palautetaan listana.

seek(paikka, mista) Kuten stdio-kirjaston *fseek()*, *seek()* asettaa tiedoston annettuun positioon *paikka*. Parametri *mista* on oletusarvoltaan nolla (absoluuttinen paikan osoitus). Muut sopivat arvot ovat 1 (suhteellinen osoitus tämänhetkisestä paikasta) ja 2 (osoitus tiedoston lopusta). Funktio ei palauta mitään arvoa.

tell() Kertoo tämänhetkisen paikan tiedostossa.

write(mjono) Kirjoittaa merkkijonon *mjono* tiedostoon. Funktio ei palauta mitään arvoa.

writelines(lista) Kirjoittaa merkkijonolistan *lista* merkkijonot tiedostoon. Merkkijonojen väliin ei lisätä (rivin) erotinmerkkejä. Funktio ei palauta mitään arvoa.

ord(s)

Palauttaa yksimerkkisen merkkijonon *s* ASCII-koodin. Funktio on aiemmin esitellyn funktion *chr()* käänteisfunktio.

pow(x, y [, z])

Palauttaa (x^y%z). Modulo *z()* on tehokkaampi laskea yhdessä potenssin kanssa kuin erikseen. Parametrien tulee olla numeerisia, tyypinmuunnokset tapahtuvat kuten aritmeettisilla operaatioilla. Jos tulos ei mahdu muunnettuun tyyppiin, aiheutetaan poikkeutus, esimerkiksi *pow(2,-1)* ja *pow(2,35000)* eivät ole sallittuja (ensimmäisessä luku 2 on kokonaisluku; kokonaisluvuille ei ole määritelty (reaalilukuista) neliöjuurta, toisessa eksponentti aiheuttaa ylivuodon).

range([alku,] loppu[, lisäys])

Palauttaa listan numeroista väliltä *[alku, loppu)*, missä vierekkäisten alkioiden erotus on suuruudeltaan *lisäys*.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([kehote])

Kysyy käyttäjältä tiedon ja palauttaa sen merkkijonona. Valinnainen parametri *kehote* on merkkijono, joka tulostetaan ennen kysymistä. Jos syötöstä luetaan tiedoston loppu (EOF), aiheutetaan *EOFError*.

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

reduce(fkt, sarja [, alustus])

Sovitetaan sarjaan sarja funktiota *fkt*, jolla *sarja* saadaan supistettua yhdeksi alkioksi. Valinnainen parametri *alustus* lisätään sarjan alkuun, jotta tyhjätkin sarjat voidaan käsitellä.

```
>>> reduce(lambda x, y: x*y, [1,2,3,4], 1)
24
```

reload(mod)

Tulkkaa ja alustaa modulin *mod* uudelleen. Modulin *mod* tulee olla kertaalleen onnistuneesti tuotu (import-komennolla) systeemiin. Palauttaa moduliolion. Katso lisätietoja monisteesta [2].

repr(olio)

Palauttaa oliosta merkkijonoesityksen, joka voidaan antaa *eval()*-funktiolle, joka palautaa sitten merkkijonoa vastaavan olion. Funktion tulos on sama kuin takaheittomerkkien (‘‘).

```
>>> repr((1,2,3))
'(1, 2, 3)'
>>> '(1,2,3)'
'(1, 2, 3)'
>>> eval(repr((1,2,3)))
(1, 2, 3)
>>> eval('(1,2,3)')
(1, 2, 3)
>>> type(repr((1,2,3)))
<type 'str'>
>>> type(eval(repr((1,2,3))))
<type 'tuple'>
```

round(x, n)

Palauttaa liukuluvun *x* pyöristettynä *n* merkitsevään desimaaliin. Oletusarvo *n*:lle on nolla. Pyöristys tehdään nollasta poispäin.

setattr(olio, nimi, arvo)

Määrittelee oliolle *olio* merkkijonon *nimi* ilmoittaman attribuutin, jonka arvo on parametri *arvo*, jos attribuutti on olemassa ja sen muuttaminen on sallittua. Lause *setattr(x, 'foobar', 123)* vastaa lausetta *x.foobar = 123*.

str(olio)

Palauttaa parametrin *olio* merkkijonoesityksen, joka 'tulostuu kauniisti'. Merkkijonoille palauttaa merkkijonon itse. Ero funktioon *repr()* on siinä, ettei *str()*-funktion palauttamasta merkkijonosta voi generoida uutta oliota *eval()*-funktiolla.

tuple(sarja)

Palauttaa monikon, jonka alkiot ovat samat kuin parametrinä annetun sarjan *sarja*.

```
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
```

type(olio)

Palauttaa parametrin olio tyypin. Standardimodulissa types on määritelty sisäänrakennetut tyypit.

```
>>> import types
>>> if type(x) == types.StringType: print "x on merkkijono."
```

vars([olio])

Ilman parametreja funktio palauttaa sanaston lokaaleista symboleista. Jos parametrina on moduli, luokka tai luokkaesiintymä tai joku muu *__dict__*-attribuutin sisältävä olio, palautetaan sanasto kyseisen olion symboleista. Palautettua sanastoa ei saa muokata, seurauksia vastaavalle symbolitaululle ei ole määritelty.

xrange([alku,] loppu[, lisäys])

Sama kuin *range()*, mutta palauttaa *xrange*-olion, joka vastaa *range()*-funktion tuottamaa listaa, mutta generoi alkion aina sitä pyydettäessä (sen sijaan, että se generoisi kaikki alkiot kerralla). Tästä on hyötyä lähinnä kun generoitava lista on pitkä ja sen kaikkia alkioita ei tulla käyttämään (esimerkiksi kun silmukan suoritus katkaistaan *break*-komennolla).

Luku 8

Modulit ja paketit

Pythonissa modulin muodostaa yksi tiedosto määrittelyineen. Tiedoston tarkentimena käytetään merkkijonoa '.py'. Python-tulkki kääntää tiedoston '.pyc'-loppuiseksi tiedostoksi, jota sitten ajetaan. Tiedoston alussa tulee ilmoittaa käytetty merkistökoodaus. Sen voi esittää emacs- tai vim-editorien tunnistamassa muodossa, tai Unicode-tunnisteella. Alla on esimerkki modulista fib, joka laskee Fibonaccin lukuja (tiedoston nimi on 'fibo.py').

```
# −∗− coding: iso−8859−15 −∗−
# Fibonaccin numerot
def fib(n): # tulostetaan Fibonaccin sarja n:ään asti
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # palautetaan Fibonaccin sarja n:ään asti
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
        return result
```

Modulia käytetään Pythonista tuomalla se sinne *import*-komennolla. Moduli voi samalla tavalla tuoda itseensä toisia moduleja.

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
>>>
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> from fibo import *
```

```
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Komento *import* on selitetty aiemmin.

Hakemistot, joissa moduleina toimivat tiedostot ovat, muodostavat paketit. Jokaisesta pakettina toimivasta tiedostosta tulee löytyä tiedosto nimeltä *__init__.py*, joka sisältää paketin alustuskoodin, joskin se voi olla tyhjä. Esimerkiksi tiedostopolun "./foo/bar/baz.py"takaa löytyvä *baz* voidaan ottaa käyttöön *import*-komennolla *import foo.bar.baz*.

8.1 Standardimodulit

Kaikki standardimodulit on esitelty kielen kehittäjän Guido van Rossumin kirjoittamassa monisteessa 'Python Library Reference' [2]. Moniste, kuten muutkin van Rossumin kirjoittamat monisteet ovat luettavissa WWW:n kautta Pythonin kotisivulta `http://www.python.org/`. Alla lueteltujen lisäksi Pythonissa on valmiita moduleita UNIX-palveluille, verkkopalveluille, salakirjoitukselle, WWW-palveluille, multimedialle ja yksittäisille koneympäristöille (Windows, Mac OS, Linux, SunOS ja SGI).

sys

Sisältää systeemille ominaiset muuttujat ja funktiot.

argv *argv* on lista komentojonoparametreista, jotka on annettu Python-ohjelmalle. Ensimmäinen alkio *sys.argv[0]* on ohjelman nimi. Jos komento on ajettu Python-tulkin parametrilla '-c', on ensimmäisen alkion arvo merkkijono '-c'. Jos parametreja ei ole annettu, on *sys.argv*:n pituus nolla.

builtin_module_names Antaa listan kaikkien niiden moduleiden nimistä, jotka on käännetty tähän tulkkiin.

exit(n) Poistutaan Python-tulkista palautusarvona *n*. Poistuttaessa aiheutetaan *SystemExit*-poikkeutus.

exitfunc Tätä ei ole asetettu, mutta käyttäjä voi määritellä tälle nimelle parametrittoman funktion, jota kutsutaan aina, kun tulkista poistutaa, paitsi jos tapahtuu järjestelmän sisäinen virhe (fatal error).

modules Palauttaa listan ladatuista moduleista.

path Lista modulien hakemistoista. Alustetaan ympäristömuuttujan *PYTHONPATH* arvolla.

ps1, ps2 Merkkijonot, jotka määrittelevät tulkin kehotteet interaktiivisessa tilassa. Oletusarvot ovat *ps1*=='»>' ja *ps2*=='...'.

stdin, stdout, stderr Standardit tietovirrat.

types

Sisältää sisäänrakennettujen tyyppien nimet. Yleensä moduli luetaan komennolla *from types import* *. Tulevissa versioissa esiintyvät uudet nimet tulevat myös loppumaan sanaan 'Type'. Moduli sisältää seuraavat tietoattribuutit:

NoneType Nollaolion (*None*) tyyppi.

TypeType Tyyppiolion tyyppi.

IntType Kokonaislukutyyppi (esim. 1)

LongType Pitkien kokonaislukujen tyyppi (esim. 1L)

FloatType Liukulukujen tyyppi (esim. 1.0).

StringType Merkkijonotyyppi (esim. 'Spam').

TupleType Monikkotyyppi (esim. (1, 2, 3, 'Spam')).

ListType Listatyyppi (esim. [0, 1, 2, 3]).

DictType tai DictionaryType Sanastotyyppi (esim. 'Pekoni': 1, 'Kinkku': 0).

FunctionType Käyttäjän määrittelemien ja lambda funktioiden tyyppi.

LambdaType Vaihtoehtoinen nimi tyypille *FunctionType*.

CodeType Koodiolioiden tyyppi (esim. *compile()*-funktion palautusarvo).

ClassType Käyttäjän määrittelemien luokkien tyyppi.

InstanceType Käyttäjän määrittelemien luokkien instanssien tyyppi.

MethodType Käyttäjän määrittelemien luokkien metodien tyyppi.

UnboundMethodType Vaihtoehtoinen nimi tyypille *MethodType*.

BuiltinFunctionType Sisäänrakennettujen funktioiden, kuten *len()* tai *sys.exit()* tyyppi.

BuiltinMethodType Vaihtoehtoinen nimi tyypille *BuiltinFunction*.

ModuleType Modulien tyyppi.

FileType Tiedosto-olioiden tyyppi (esim. sys.stdout).

XRangeType *xrange()*-funktion palauttaman olion tyyppi.

TracebackType Traceback-olion tyyppi(esim. *sys.exc_traceback*).

FrameType Kehysolioiden tyyppi.

traceback

Traceback-oliot ja niiden käyttö.

pickle

Funktiot Python-olioiden muuntamiseksi tavujonoksi ja takaisin.

shelve

Sanaston lailla käyttäytyvä tietosäilö (tietokanta) olioille.

copy

Sisältää funktiot olioiden kopioimiseen.

copy.copy(n) Muodostaa oliosta *n* yksinkertaisen (shallow) kopion, se ei tee kopioita olion sisältämien viittausten kohteista, vaan kopioi viittaukset.

copy.deepcopy(n) Muodostaa oliosta *n* syvän (deep) kopion, eli tekee olion sisältämien viittausten kohteista myös kopiot. Jos tapahtuu virheitä, aiheutetaan *copy.error*-poikkeutus.

Tämä versio ei kopioi moduleita, luokkia, funktioita, metodeja, pino-olioita, tiedosto- tai socket-olioita, ikkunoita, taulukoita (array) eikä vastaavia tyyppejä.

marshal

Modulin avulla voidaan kirjoittaa ja lukea yksinkertaisia Python-olioita levyltä tai levylle. Kannattaa käyttää ennemmin pickle- ja shelve-moduleita.

imp

Sisältää liittymän *import*-komennon toimintojen toteuttamiseen.

__builtin__

Sisältää sisäänrakennetut funktiot, jotka on esitelty aikaisemmin.

8.2 Merkkijonojen käsittelyyn tarkoitetut modulit

string

Sisältää yleisimmät merkkijonojen käsittelyfunktiot. Listassa on annettu kuvaus vain erikoisimmista funktioista ja vakioista.

digits Merkkijono '0123456789'.

hexdigits Merkkijono '0123456789abcdefABCDEF'.

letters Merkkijono, joka on kahden seuraavan katenaatio.

lowercase Systeemistä riippuva merkkijono, yleensä 'abcdefghijklmnopqrstuvwxyz'. Sisällön muuttamisesta aiheutuu ongelmia funktioiden *upper()* ja *swapcase()* käytössä.

octdigits Merkkijono '01234567'.

uppercase Systeemistä riippuva merkkijono, yleensä 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. Sisällön muuttamisesta aiheutuu ongelmia funktioiden *lower()* ja *swapcase()* käytössä.

whitespace Merkkijono, joka sisältää systeemin välimerkit. Muuttamisesta aiheutuu ongelmia funktioiden *strip()* ja *split()* käytössä.

Modulissa määritellyt funktiot ovat:

atof(s) Muuntaa merkkijonon s liukuluvuksi.

atoi(s [, kanta)] Muuntaa merkkijonon s kokonaisluvuksi annetussa kannassa kanta (8=oktaali, 10=desimaali, 16=heksadesimaali).

atol(s[, kanta)] Muuntaa merkkijonon s pitkäksi kokonaisluvuksi annetussa kannassa kanta (8=oktaali, 10=desimaali, 16=heksadesimaali).

expandtabs(s, tab_koko) Korvaa merkkijonon TAB-merkit tab_koko välilyönnillä.

find(s, ali[, alku)] Palauttaa alimman indeksin merkkijonossa s, joka ei ole alku:a pienempi, josta alimerkkijono ali löytyy. Palauttaa -1, jos alimerkkijonoa ei löydy.

rfind(s, ali[, alku)] Kuten find(), mutta etsii suurimman indeksin, josta alimerkkijono löytyy.

index(s, ali[, alku)] Kuten find(), mutta aiheuttaa poikkeutuksen ValueError, jos alimerkkijonoa ei löydy.

rindex(s, ali[, alku)] Kuten rfind(), mutta aiheuttaa poikkeutuksen ValueError, jos alimerkkijonoa ei löydy.

count(s, ali[, alku)] Palauttaa alimerkkijonon ali ei päällekkäisten esiintymien määrän merkkijonossa s.

lower(s) Muuttaa merkkijonon merkit pieniksi.

split(s) Palauttaa listan merkkijonon s whitespace-merkeillä erotelluista sanoista.

splitfields(s, erotin) Palauttaa listan merkkijonon s merkkijonolla erotin erotelluista sanoista.

join(sana_lista) Yhdistää listan (tai monikon) sana_lista sanat merkkijonoksi erotellen ne välilyönnillä.

joinfields(sana_lista, erotin) Sama kuin edellinen, mutta käyttää erottimena sanojen välissä merkkijonoa erotin.

strip(s) Poistaa tarpeettomat whitespace-merkit merkkijonon s alusta ja lopusta.

swapcase(s) Vaihtaa merkkijonon s isot kirjaimet pieniksi ja pienet isoiksi.

upper(s) Muuttaa merkkijonon s kirjaimet isoiksi.

ljust(s, leveys), rjust(s, leveys), center(s, leveys) Sisentävät tai keskittävät merkkijonon s nimensä mukaan leveys merkkiä leveään kenttään. Merkkijonoa ei katkaista, joten tuloksen pituus voi olla suurempikin kuin leveys.

zfill(s, leveys) Merkkijonon s sisältämän numeron eteen tulee tarvittava määrä nollia kentän leveyden leveys täyttämiseksi.

re

Säännöllisten lauseiden käsittely- ja korvausfunktiot.

struct

Tietueiden tekoon tarvittavat funktiot. Tarvitaan lähinnä rakennettaessa Python-kielen laajennuksia C-kielellä.

8.3 Muita moduleita

math

Sisältää seuraavat matemaattiset funktiot: acos(x), asin(x), atan(x), atan2(x, y), ceil(x), cos(x), cosh(x), exp(x), fabs(x), floor(x), fmod(x, y), frexp(x), hypot(x, y), ldexp(x, y), log(x), log10(x), modf(x), pow(x, y), sin(x), sinh(x), sqrt(x), tan(x), tanh(x).

frexp()- ja modf()-funktioilla on erillaiset parametrit ja palautusarvot kuin C-kielen vastaavilla. hypot()-funktio ei ole määritelty kaikissa laitteissa, sillä se ei kuulu standardi C-kieleen.

Modulissa on myös määritelty matemaattiset vakiot e ja pi (vastaaville nimille).

random

Sisältää mm. pseudo-satunnaislukuja palauttavan funktion, joka on C-kielen rand()-funktion kaltainen. Sisältää myös muita funktioita satunnaisten otantojen tai operaatioiden tekoon.

array

Modulin avulla voidaan määritellä ja käyttää kompakteja ja tehokkaita taulukoita.

os

Sisältää yleiset käyttöjärjestelmän palvelut toteuttavat funktiot ja muuttujat.

time

Kellonajan ja päiväyksen selvittämiseen tarvittavat funktiot.

asctime(t) Konvertoi yhdeksänalkioisen monikon t (jonka esim. gmtime() on palauttanut) merkkijonoksi (jonka loppuun ei tule rivinvaihtoa.

clock() Tämänhetkinen CPU-aika sekunteina.

ctime(s) Konvertoi lokaalin ajan sekunneista (s sekunteina epookista).

gmtime(s) Palauttaa UTC ajan yhdeksänalkioisen monikon sekunneista s (s sekunteina epookista). Monikon alkiot ovat vuosi, kuukausi (1-12), päivä (1-31), tunnit (0-23), minuutit (0-59), sekunnit (0-59), viikonpäivä (0-6, 0 on maanantai), Juliaaninen päivä (1-366) ja kesäajan ilmoittava lippu (aina nolla!).

localtime(s) Palauttaa yhdeksänalkioisen monikon, jossa on paikallinen aika.

mktime(t) localtime()-funktion käänteisfunktio. t on täysi yhdeksänalkioinen monikko, sillä kesäaika lipun arvo huomioidaan!

sleep(s) Ohjelman suoritus pysäytetään *s* sekunniksi.

time() Palauttaa UTC ajan sekunteina epokista (reaalilukuna).

Modulin muuttujat ovat:

altzone kesäaikamuutos sekunteina (länsi positiivinen, itä negatiivinen). Käytä vain, jos daylight!=0.

daylight !=0, jos kesäaika määritelty

timezone aikavyöhyke sekunteina ei-kesäajassa (länsi positiivinen, itä negatiivinen)

tzname aikavyöhykkeen nimi

pdb

Pythonilla toteutettu Python-debuggeri. Modulin kuvaus löytyy lähteestä 'Python Library Reference' [2].

Luku 9

Oliot ja luokat

Pythonin luokat ovat sekoitus C++:n ja Modula-3:n luokista. Niiden määrittely ja käyttö on tehty mahdollisimman selkeäksi. Kuten moduleissa, luokissakin Python luottaa ohjelmoijansa käytöstapoihin – luokkien attribuutteja ei ole suojattu muuntamiselta C++:n tapaan. Pythonin luokat voivat periä ominaisuuksia yhdeltä tai useammalta luokalta, peritty luokka voi ylimääritellä edeltäjänsä metodeita ja luokan metodi voi kutsua ylemmän luokan samannimisiä metodeja.

Kaikki luokan jäsenet (myös tieto) ovat julkisia (C++:ssa public), ja jäsenfunktiot ovat virtuaalisia. Jos kuitenkin jäsenen nimen edessa on kaksi alleviivausmerkkiä, muokkaa Python nimen niin, ettei se ole sellaisenaan näkyvissä luokan tai olion piirteitä listatessa. Olion alustus kirjoitetaan metodiin *__init__(self, ...)*. Mitään lyhennysmerkintää olion omiin jäseniin viittaamiseen ei ole; viittaus olioon tulee jäsenfunktion ensimmäisenä parametrina (parametrin nimeä ei ole määrätty, mutta yleinen tapa on käyttää nimeä 'self'). Koska Pythonissa kaikki tietotyypit ovat olioita, ovat luokatkin olioita. Kuitenkaan sisäänrakennetut tyypit eivät voi toimia itsemääritellyn luokan kantaluokkina. Myös suurimman osan sisäänrakennetuista operaattoreista voi ylimääritellä luokan jäsenille.

Pythonin sisäänrakennetut tyypit ovat siis olioita, mutta kaikki eivät ole luokkia, esimerkiksi kokonaisluvut, listat ja tiedostot eivät ole luokkia. Kuitenkin kaikki tyypit sisältävät ominaisuuksia, joiden vuoksi niitä voidaan kutsua olioiksi.

Oliot ovat yksittäisiä, ja niihin voi sitoa useita eri nimiä eri näkyvyysalueista (aliasing). Tämä ominaisuus on ensin hieman hämmentävä, mutta tajuamisen jälkeen erittäin tehokas ja hyödyllinen, aliakset käyttäytyvät hieman kuten osoittimet tai viittaukset (references). Olion välittäminen funktiolle tai metodille on näinollen tehokasta ja halpaa, ja funktion olioon tekemät muutokset näkyvät takaisin käyttäjälle.

Luokan esittely on yksinkertaisimmillaan seuraavanlainen:

```
1 class LuokanNimi:
2    <lause-1>
3    .
4    .
5    .
6    <lause-N>
```

Luokkaan kuuluvat määrittelyt ovat siis sisennettyinä, kuten funktioiden määrittelyt ja lohkot yleensä. Luokan määrittely täytyy ajaa ennenkuin sitä voidaan käyttää. Luokalla on oma sisäinen näkyvyysalueensa.

Kun luokan määrittelystä poistutaan luodaan luokkaolio, sille nimiavaruus ja sidotaan luokka luokan nimeen *LuokanNimi*.

Luodaanpas pieni luokka:

```
class Luokka:
    def __init__(self, arvo=1234):
        self.i=arvo
    def f(self):
        return 'Terve, Maailma!'
```

Nyt Luokka.i ja Luokka.f ovat luokan olioiden attribuutteja, joista ensimmäinen palauttaa kokonaisluvun ja toinen funktion. Luokasta luodaan olio seuraavasti (sulkeissa olisi mahdolliset alustusparametrit, mutta niitä ei nyt ole).

```
x=Luokka()
```

Nyt meillä on luokan Luokka instanssi, johon lokaali muuttuja (nimi) x viittaa. Sitä voidaan käyttää tuttuun tapaan erottamalla instanssin ja attribuutin nimi toisistaan pisteellä. Alla i on tietoattribuutti ja f on metodi. Tietoattribuutit peittävät metodit.

```
>>> x.i
1234
>>> x.f()
Terve, Maailma!
```

Käyttäjän määrittelemille luokille voidaan määritellä joitain metodeja, jotka ylimäärittelevät operaatioita kuten aritmeettiset operaatiot, indeksointi, paloittelu tai alustus. Alla on lueteltu näistä osa, tarkempi kuvaus löytyy monisteesta 'Python Reference Manual' [2].

__init__(self, args...) Kutsutaan oliota luotaessa. Argumentit ovat samat, jotka annetaan luokan nimen jälkeen oliota luotaessa. Perityn luokan tulee kutsua isätäluokkansa __init__-metodia instanssinsa oikean alustamisen varmistamiseksi.

__del__(self) Kutsutaan olion hävitessä. Perittyjen luokkien __del__-metodia on syytä kutsua kunnon hävityksen varmistamiseksi. Huomaa, ettei del x välttämättä kutsu tätä metodia, paitsi jos x on viimeinen viittaus tuhottavaan olioon.

__repr__(self) Kutsutaan sisäistä funktiota repr() kutsuttaessa ja muunnettaessa olio merkkijonoksi ‘‘-merkeillä (takaheittomerkit). __repr__()-metodilla tuotetusta merkkijonosta voidaan luoda uusi olio eval()-funktiolla.

__str__(self) Kutsutaan sisäistä funktiota str() kutsuttaessa ja print-komentoa suoritettaessa.

__cmp__(self, other) Kutsutaan kaikkien vertailujen yhteydessä. Jos self<other, tämän tulee palauttaa -1, jos self==other tulee palauttaa 0 ja jos self>other, tulee palauttaa 1.

__add__(self, other) Yhteenlaskuoperaatio +

__sub__(self, other) Vähennyslaskuoperaatio -

__mul__(self, other) Kertolaskuoperaatio *

__div__(self, other) Jakolaskuoperaatio /

__divmod__(self, other) divmod() operaatio

__pow__(self, other) Potenssiinkorotusoperaatio pow()

__len__(self) Kutsutaan kutsuttaessa sisäistä funktiota len(). Palauttaa olion 'pituuden' (joka on >=0). Jos olion pituus ==0, vastaa se Boolean-arvoa epätosi.

__getitem__(self, avain) Kutsutaan viitattaessa olion alkioon hakasulkeissa, siis self[avain]. Negatiiviset avaimet on huomioitava itse.

__setitem__(self, avain, arvo) Asettaa olion alkiolle avain arvon arvo.

__delitem__(self, avain) Metodi sarjaolion alkion self[avain] tuhoamiseksi.

__getslice__(self, i, j) Kuten yhdelle alkiolle, tällä luetaan viipaleen (alkiosarjan) arvo. Puuttuva i tai j korvataan vastaavasti nollalla tai arvolla len(self). Päinvastoin kuin __getitem__-metodilla, on negatiiviseen i:hin tai j:hin lisätty len(self).

__setslice__(self, i, j, sarja) Asetetaan viipaleen arvo.

__delslice__(self, i, j) Tuhotaan viipale.

9.1 Perintä

Luokalla voi myös olla yksi tai useampia kantaluokkia. Nämä ilmoitetaan luokan määrittelyssä luokan nimen perässä sulkeissa pilkuilla eroteltuina.

Johdettu luokka ylimäärittelee kantaluokkansa samannimiset metodit. Koska metodit ovat virtuaalisia, voi kantaluokassa tapahtuva saman luokan toisen metodin kutsu itseasiassa suorittaa johdetun luokan ylimäärittelevää metodia. Jos kantaluokkia on useampia kuin yksi, suoritettavaa metodia etsitään syvähakuna vasemmalta oikealle.

Alla on esimerkkinä moduli firma.py, joka määrittelee kantaluokan tyontekija ja siitä perityn luokan pomo.

```
1  # −∗− coding: iso−8859−15 −∗−
2  class Tyontekija:
3    def __init__(self, nimi, osoite):
4      self.nimi=nimi
5      self.osoite=osoite
6    def __str__(self):
7      return self.nimi+'\n'+self.osoite
8  class Pomo(Tyontekija):
9    def __init__(self, nimi, osoite, alaiset=None):
10     tyontekija.__init__(self,nimi,osoite) # Huomaa kantaluokan metodin kutsuminen
11     if alaiset!=None:
12         self.alaiset=alaiset
13     else:
14         self.alaiset=[]
15   def tulosta_alaiset(self):
16     for a in self.alaiset:
17       print a, '\n'
```

Seuraavaksi on esimerkkiajo, jossa luodaan kolme tyontekija-oliota ja yksi pomo-olio, ja tulostetaan näiden arvot:

```
1  >>> import firma
2  >>> a1=firma.Tyontekija('Jaska','Kuja 1')
```

```
>>> a2=firma.Tyontekija('Kale','Kuja 2')
>>> p1=firma.Pomo('Iso-G','Linna',
...     [a1,a2,firma.Tyontekija('Erkki','Stönö')])
>>> print a1
Jaska
Kuja 1
>>> print a2
Kale
Kuja 2
>>> print p1
Iso-G
Linna
>>> p1.tulosta_alaiset()
Jaska
Kuja 1

Kale
Kuja 2

Erkki
Stönö

>>>
```

Luku 10

Laajennettavuus

Yksi Pythonin parhaita puolia on kielen laajennettavuus. Kieltä voi joko laajentaa Python-kielisillä tai C/C++-kielisillä moduleilla. Tällöin itse Pythonilla voidaan tehdä kehysohjelma, ylemmän tason ohjelma, jossa tarvitaan korkeaa kieltä, ja nopeutta vaativat rutiinit voidaan toteuttaa C-kielellä. Python-tulkin lähdekoodin mukana tulee ohjeet ja esimerkkejä laajennuksien tekoon, jota on myös selitetty monisteessa 'Python Extension Manual' [1], joka on luettavissa WWW:in kautta Pythonin kotisivulta (katso Johdanto kappaletta). Myös omista C-kielisitä ohjelmista voi käyttää Python-tulkkia. Laajennuksia varten Python-kielessä on oma pieni API. Laajennukset voivat olla joko tulkkiin sisäänrakennettuja tai dynaamisia.

Toinen laajennuksista saavutettava hyöty on tiedon suojaus. Jos jonkin luokan attribuutteja todella täytyy suojata, voidaan luokka kirjoittaa C-kielisenä laajennuksena, ja esitellä siitä Pythoniin näkymään vain ne attribuutit, jotka saavatkin näkyä.

Kirjallisuutta

[1] Guido van Rossum: *Extending and embedding the Python interpreter*, Report CS-R9527, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, huhtikuu 1995.
URL `http://www.python.org/doc/ext/`

[2] Guido van Rossum: *Python library reference*, Report CS-R9524, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, huhtikuu 1995.
URL `http://www.python.org/doc/lib/`

[3] Guido van Rossum: *Python reference manual*, Report CS-R9525, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, huhtikuu 1995.
URL `http://www.python.org/doc/ref/`

[4] Guido van Rossum: *Python tutorial*, Report CS-R9526, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, huhtikuu 1995.
URL `http://www.python.org/doc/tut/`