Metaohjelmointi Python-kielellä

Mikko Koho

Seminaaritutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 18. joulukuuta 2014

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Mikko Koho

Työn nimi — Arbetets titel — Title

Metaohjelmointi Python-kielellä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaaritutkielma	18. joulukuuta 2014	15

Tiivistelmä — Referat — Abstract

Tämä seminaaritutkielma käsittelee Python-ohjelmointikielen tarjoamia mahdollisuuksia metaohjelmointiin. Metaohjelmointi tarkoittaa sellaisten ohjelmien tekemistä, jotka tekevät uusia ohjelmia. Metaohjelmointiin kuuluu myös ohjelman suoritusaikainen itsensä tarkastelu ja muokkaus.

Python on dynaaminen ohjelmointikieli ja tarjoaa hyvät mahdollisuudet tarkastella ja muokata ohjelmaa suoritusaikana. Luokkia ja olioita voidaan muokata vapaasti, uusia luokkia voidaan luoda suoritusaikana. Uutta ohjelmakoodia voidaan kääntää ja ajaa osana ohjelman suoritusta.

Pythonin standardikirjastossa on monia metaohjelmointia tukevia moduuleita kuten ast, dis, inspect ja parser.

ACM Computing Classification System (CCS):
Software and its engineering → Language features
Software and its engineering → Multiparadigm languages
Software and its engineering → Compilers

Avainsanat — Nyckelord — Keywords
Python, ohjelmointi, metaohjelmointi

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Metaohjelmoinnilla tarkoitetaan klassisen määritelmän mukaan sellaisen tietokoneohjelman tekemistä, joka kirjoittaa uusia tietokoneohjelmia [HB12, s. 6]. Tämä on kuitenkin melko yksinkertaistettu määritelmä, eikä metaohjelmointia ole helppo määritellä tarkasti. Toinen yleinen määritelmä esittää metaohjelmoinnin olevan ”tietokoneohjelma, joka manipuloi toisia ohjelmia suoritusaikana” [HB12, s. 7].

Tärkeä osa metaohjelmointia on ohjelman suoritusaikainen tilansa tarkastelu ja muuntelu eli *reflektio*. Reflektioon kuuluu käsite *introspektio*, jolla tarkoitetaan suoritusaikaista muistissa olevien olioiden tarkastelua [Pil04].

Tässä seminaarityössä tarkastellaan Python-ohjelmointikielen tarjoamia työkaluja metaohjelmointiin. Luvussa 2 käydään läpi Python-kielen perusteita. Luvussa 3 käsitellään metaohjelmointia Python-kielellä. Metaohjelmoinnista tarkastellaan lähinnä suoritusaikaista metaohjelmointia.

2 Python-ohjelmointikieli

Ensimmäinen Python-kielen versio on julkaistu 1991. Pythonin suosio on kasvanut tasaisesti, ja se on nykyään käytetyin kieli ohjelmoinnin perusteiden opetukseen Yhdysvaltojen yliopistoissa [Guo14]. Python-kielestä on nykyään käytössä eri versioita. Python 2.7 on edelleen melko suosittu, vaikka versio 3 on julkaistu jo 2008. Versio 3 ei ole yhteensopiva aiempien versioiden kanssa. Python 3:n yleistymistä on hidastanut se, että jotkut suositut kirjastot ja sovelluskehykset käyttävät edelleen Python 2:ta eivätkä ole siirtyneet versioon 3.

Python-ohjelmakoodia voidaan kääntää useilla eri kääntäjillä [Mar06, s. 5]. Käytetyin kääntäjä on CPython (Classic Python), joka kääntää alkuperäisen koodin Python-tavukoodiksi. Muita suosittuja kääntäjiä ovat Java-tavukoodiksi kääntävä Jython sekä IronPython, joka kääntää Python-koodia .NET-ympäristön käyttämäksi CIL-tavukoodiksi. PyPy on Python-kielellä toteutettu useissa eri ympäristöissä toimiva suoraan konekielelle koodia kääntävä suoritusaikainen (just-in-time) kääntäjä.

CPythonilla käännettyä tavukoodia voidaan ajaa C-kielellä toteutetulla

tulkilla [Mar06, s. 22] eli virtuaalikoneella. Pythonin standardikirjasto on toteutettu osittain C:llä ja osittain Pythonilla.

Seminaarityön esimerkit toimivat sekä Python 2.7:llä että Python 3:lla. Esimerkeissä ohjelman tulosteet on kirjoitettu kommenttiriveinä tulosteen tuottavan rivin jälkeen.

Tässä luvussa käydään läpi Python-kielen perusteita ja metaohjelmoinnin kannalta olennaisia asioita.

2.1 Syntaksi

Python-ohjelma koostuu loogisista riveistä, jotka ovat yhden tai useamman fyysisen rivin mittaisia [Mar06, s. 33]. Loogisten rivien päättämiseen ei käytetä mitään merkkiä. Rivien sisennyksen perusteella erotetaan ohjelmakoodin lohkot toisistaan. Suositeltu tapa sisentää on käyttää ensimmäisen tason sisentämiseen 4 välilyöntiä ja seuraavaan 8 ja niin edelleen [VWC13].

Pythonissa on 30 avainsanaa (keyword), jotka ovat kielen varattuja sanoja. Näitä ovat esimerkiksi funktio `print` ja kielen rakenteissa käytetyt sanat kuten `if`, `and` ja `class`. Pythonin standardikirjasto koostuu sisäänrakennettujen funktioiden lisäksi kokoelmasta eri tarkoituksiin soveltuvia moduuleita (module), jotka pitää tarvittaessa tuoda erikseen osaksi suoritettavaa ohjelmaa komennolla `import`.

2.2 Muuttujat

Python on dynaamisesti tyypitetty kieli. Muuttujien arvon tyyppiä ei tarvitse eksplisiittisesti määrittää, vaan tyyppi määräytyy sen perusteella minkälainen arvo muuttujaan sijoitetaan. Python on lisäksi dynaaminen kieli, eli muuttujan sisältämää arvoa voidaan muokata vapaasti ajon aikana. Muuttujan arvoa voidaan vaihtaa sijoittamalla siihen uusi arvo. Uuden arvon ei tarvitse olla saman tyyppinen kuin muuttujan vanha arvo, vaan muutujan tyyppiä voidaan vaihtaa sijoittamalla siihen eri tyyppinen arvo. Listaus 1 sisältää yksinkertaisen esimerkin Python-kielen syntaksista. Rivillä 1 asetetaan muuttujan `a` arvoksi merkkijono "`Hello world!`", joka tulostetaan rivillä 2. Rivillä 5 tulostetaan muuttujassa `a` olevan merkkijonon pituus.

```
1  a = 'Hello world!'
2  print(a)
3  # Hello world!
4
5  print(len(a))
6  # 12
```

Listaus 1: Yksinkertainen esimerkki Python-kielen syntaksista.

2.3 Tietotyypit

Pythonissa kaikki arvot, muuttujat ja funktiot ovat olioita. Olion tyyppi määrittää, mitä metodeja ja ominaisuuksia olio tarjoaa. Osa olioista on muuttumattomia (immutable) ja osa muutettavia (mutable). Python-kielessä ei ole erikseen vakioita, mutta käytäntönä on käyttää muuttujan nimessä pelkästään isoja kirjaimia, jos arvoa ei ole tarkoitus muuttaa.

Pythonin sisäänrakennettuja tietotyyppejä ovat muun muuassa numeeriset `int` (kokonaisluku) ja `float` (liukuluku), sekvenssityypit `list` (lista), `str` (merkkijono) ja `tuple` (monikko), joukko `set`, sanakirja `dict` sekä tiedosto `file`. Sanakirja-tyyppi tunnetaan joissain ohjelmointikielissä hajautustauluna. Hajautustaulu onkin ainakin CPythonin sisäinen toteutus sanakirjasta, mutta muitakin mahdollisia toteutustapoja on. Standardikirjastoon kuuluvista moduuleista löytyy lisää tietotyyppejä kuten aikatyyppi `datetime` ja taulukko `array`.

Sekvenssit ovat iteroitavia (iterable) olioita, eli ne kykenevät palauttamaan jäseniään yksi kerrallaan. Iteroitavia olioita ovat myös muut oliot, joissa on toteutettu jäseniä palauttava `__iter__` tai `__get_item__` -metodi. Iteroitavia olioita voidaan käyttää suoraan osana esimerkiksi `for` -toistolauseissa. kuten esimerkiksi lauseessa
`for x in [1,2,3]: print(x)`

2.4 Kehitelmät

Listakehitelmä (list comprehension) on matemaatiikan joukkojen määrittelyn merkintätavan pohjalta luotu syntaksi listan määrittelyyn olemassa olevien listojen pohjalta. Pythoniin on kehitetty myös listakehitelmän syntaksiin perustuvat joukkokehitelmä (set comprehension) ja sanakirjakehitelmä (dictionary comprehension). Nämä kaikki kehitelmät ovat helppoja tapoja luoda lista-, joukko- tai sanakirjaolioita.

Esimerkkejä listakehitelmän käytöstä sekä joidenkin Python-kielen funktioiden käytöstä on listauksessa 2. Rivillä 1 käytetty funktio `range` palauttaa Python 2:ssa listan ja Python 3:ssa generaattoriolion, jota voidaan käyttää listan tapaan. Rivillä 4 luodaan lista, joka sisältää lukujen neliöitä. Rivillä 8 suodatetaan edellisellä rivillä luodusta listasta pois kaikki alkiot, joiden merkkijonoesitykset eivät koostu pelkästään kirjaimista. Rivillä 12 lasketaan alkulukuja väliltä 2–50. Funktio `all` tarkastaa kaikkien listan (tai muun iteroitavan olion) totuusarvon. Pythonissa kaikki sisäänrakennettujen tietotyyppien oliot voidaan evaluoida totuusarvoina, jolloin lukujen tapauksessa aina luku ”0” evaluoituu epätodeksi ja muut luvut todeksi.

2.5 Luokat ja oliot

Python on olio-ohjelmointikieli, joka ei kuitenkaan pakota käyttämään olio-ohjelmointia ohjelmien toteutuksessa.

Pythonissa myös luokat ovat olioita. Luokilla on joitain piirteitä kuten muodostin (constructor) metodissa `__init__`, jotka mahdollistavat niiden käyttämisen luokkina. Luokka voi periä yhden tai useamman luokan. Luokka määritellään lausekkeella `class ClassName(InheritedClass , AnotherInherited)`. Python 2:ssa luokan tulisi periä ainakin `object`, jolloin se muodostaa niin sanotun uudentyyppisen luokan [Mar06, s. 81]. Python 2.2:ssa luokkien rakenne muuttui ja uusi rakenne toteutettiin `object`-luokkaan, jota käytetään Python 3:ssa oletuksena. Uudentyyppisten luokkien *metaluokka* on aina `type` [Mar06, s. 117]. Metaluokkaa käytetään muodostimena luotaessa uusi luokkaolio [Met14]. Metaluokkia voi määritellä itse, mutta tälle on harvoin tarvetta.

Kahdella alaviivalla metodin nimen alussa ja lopussa merkitään luokan

```
1  print(range(10))
2  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3
4  print( [x**2 for x in range(11)] )
5  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
6
7  some_list = [1, 4, 7, 'foo', 12, 'bar']
8  print( [x for x in some_list if str(x).isalpha()] )
9  # ['foo', 'bar']
10
11 print(  # Alkulukuja
12   [x for x in range(2, 50) if all(
13     [x % y for y in range(2, x-1)])])
14 # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Listaus 2: Esimerkki listakehitelmistä ja funktion range käytöstä.

”maagisia” metodeita (magic methods) ja attribuutteja [Mar06, s. 104]. Näillä erikoisilla metodeilla on jokin erityinen käyttötarkoitus Pythonissa. Niiden toimintaa ja käyttöä on selitetty Pythonin dokumentaatiossa [Ket12]. Osa ”maagisista” metodeista on sellaisia, että Python generoi ne olion luomisen yhteydessä, jolloin ne sisältävät jotain metatietoa itse oliosta. Monet metodeista ovat sellaisia, että niitä määrittelemällä tai muokkaamalla jo määriteltyjä voidaan vaikuttaa olion toimintaan ohjelmassa [Ket12, Pil09]. Esimerkiksi muokkaamalla luokan `__cmp__(self, other)`-metodia, voidaan ylikirjoittaa kahden luokasta muodostetun olion suuruusvertailu omalla metodilla. Samoin aritmeettisille operaattoreille on kullekin omat metodit.

Moniperinnästä on esimerkki listauksessa 3. Esimerkissä määritellään ensin luokka `MyClassA`, joka sisältää metodin `a`. Sen jälkeen riviltä 5 alkaen määritellään luokka `MyClassB`, joka sisältää metodin `b`. Rivillä 9 määritellään luokka `MyClassC`, joka perii molemmat aiemmin määritellyt luokat ja sisältää metodin `c`. Metodi `c` kutsuu yläluokkien metodeita `a` ja `b`. Rivillä 16 kutsutaan

luokasta `MyClassC` luodun olion metodia `c`, joka tulostaa rivit ”foo” ja ”bar”.

```
1  class MyClassA(object):
2    def a(self):
3      print('foo')
4
5  class MyClassB(object):
6    def b(self):
7      print('bar')
8
9  class MyClassC(MyClassA, MyClassB):
10   """Moniperija"""
11   def c(self):
12     self.a()
13     self.b()
14
15 olio = MyClassC()
16 olio.c()
17 # foo
18 # bar
```

Listaus 3: Esimerkki luokkien moniperinnästä.

Jos luokka perii useammalta yläluokalta attribuutteja, joilla on sama nimi, perintäjärjestys vaikuttaa siihen, mikä näistä periytyy uudelle luokalle. Yläluokkien metodeja voidaan kutsua `super`-funktiolla.

3 Pythonin metaohjelmointipiirteet

Metaohjelmointi on Pythonilla luontevaa, koska olioita voidaan tutkailla ja yleensä muokata suoritusaikana vapaasti, ja niistä saadaan paljon metatietoa Pythonin peruskirjaston työkaluilla.

3.1 Reflektio

Tarkasteltaessa suoritusaikana oliota, sen luokka saadaan kutsumalla `type`:ä niin, että parametrina on olio, jonka luokka halutaan selvittää [Pil04].

Funktio `dir` palauttaa listan olion attribuuteista, joihin kuuluu myös olion metodit. Funktio `getattr` ottaa parametrina merkkijonon ja palauttaa parametrin nimisen attribuutin. Tällöin voidaan esimerkiksi kutsua metodeja oliosta, jonka rakennetta ei tunnetta vielä käännösvaiheessa [Pil04]. Funktiolla `isinstance` voidaan tarkistaa, onko joku olio tietyn tyyppinen.

Olion lyhyen sanallisen kuvauksen saa suoritusaikana haettua sen `__doc__` -attribuutista. Kuvaus on koodissa oliolle annettu *docstring*-kuvailu, joka on Pythonissa tyypillinen tapa kuvailla tekstinä ohjelman komponentteja.

Olioita, luokkia ja funktioita voidaan muokata suoritusaikana melko vapaasti. Esimerkki standardikirjaston `dir` funktion korvaamisesta omalla funktiolla on listauksessa 4. Esimerkin oma funktio `x` palauttaa sille annetut argumentit tekemättä niille mitään. Tämänkaltaisesta kirjastojen ja moduuleiden osien suoritusaikaisesta muokkaamisesta käytetään nimeä ”monkey patching”.

```
1  import __builtin__
2
3  def x(*args):
4    return args
5
6  print(dir(__builtin__))
7  # ['ArithmeticError', 'AssertionError', ... ]
8
9  __builtin__.dir = x
10
11 print(dir(__builtin__))
12 # (<module '__builtin__' (built-in)>,)
```

Listaus 4: Standardikirjaston funktion korvaaminen omalla funktiolla.

Pythonin standardikirjaston inspect-moduuli tarjoaa työkaluja olioiden tilan tutkimiseen ohjelman suoritusaikana. Moduuli sys tarjoaa työkaluja käytetyn Python-tulkin ja käyttöjärjestelmän tarkasteluun.

Listauksessa 5 tarkastellaan inspect-moduulilla aiemmin listauksessa 3 luotua oliota nimeltä `olio`. Rivillä 1 tulostetaan olion `olio` `c`-metodin lähdekoodi. Rivellä 6 ja 9 selvitetään onko parametrinä annettu olio funktio tai metodi. Rivillä 12 käytetty `inspect.getmro` palauttaa luokan kaikki yläluokat perintäjärjestyksessä.

```
1  print(inspect.getsource(olio.c))
2  #      def c(self):
3  #            self.a()
4  #            self.b()
5
6  print(inspect.isfunction(olio.c))
7  # False
8
9  print(inspect.ismethod(olio.c))
10 # True
11
12 print(inspect.getmro(type(olio)))
13 # (<class '__main__.MyClassC'>,
14 #  <class '__main__.MyClassA'>,
15 #  <class '__main__.MyClassB'>,
16 #  <type 'object'>)
```

Listaus 5: Python-olion tarkastelua inspect-moduulilla. Tarkasteltava olio on luotu aiemmin listauksessa 3.

3.2 Introspektion muokkaaminen

Todennäköisesti harvoin hyödyllinen ominaisuus on Pythonissa on se, että voidaan suoritusaikana oliokohtaisesti vaikuttaa siihen, miten olioiden

introspektio toimii.

Metaluokkien avulla siis luodaan luokkia. Metaluokan metodeilla `__instancecheck__(self, instance)` ja `__subclasscheck__(self, subclass)` voidaan vaikuttaa siihen miten luokasta muodostettujen olioiden tarkastelu käyttäytyy [Ket12, Pyt]. Näistä ensimmäistä käytetään yliajamaan (override) Pythonin sisäänrakennetun funktion `isinstance` toimintaa, joka kertoo onko instanssi muodostettu tietystä luokasta. Jälkimmäinen yliajaa sisäänrakennetun funktion `issubclass` toimintaa ja kertoo onko luokka toisen luokan aliluokka.

3.3 Dynaamiset luokat

Luokkia voidaan muokata ohjelman suoritusaikana, jolloin vaikutus ulottuu kaikkiin jo luotuihin luokan instansseihin. On myös mahdollista luoda uusia luokkia dynaamisesti ohjelman suoritusaikana `type`-funktion avulla, joka toimii myös metaluokkana [Met14]. Kutsumalla `type`:ä kolmella parametrilla saadaan paluuarvona luokka. Parametrit ovat järjestyksessä luokan nimi, perittävät luokat ja sanakirja-tyyppinen muuttuja luokan sisällöstä [Met14].

Esimerkki luokan luomisesta dynaamisesti ajon aikana on listauksessa 6. Esimerkin rivillä 1 määritellään funktio `dynamic_c`. Rivillä 5 luodaan `type`-funktiolla uusi luokka. Parametreinä `type`-funktiolle annetaan luokan nimi, perittävät luokat sekä luokan attribuutit. Perittävät luokat `MyClassA` ja `MyClassB` on määritelty listauksessa 3. Rivillä 13 luodaan olio tehdystä luokasta ja seuraavalla rivillä kutsutaan sen metodia `c`.

3.4 Koodin kääntäminen suoritusaikana

Pythonissa on sisäänrakennetut funktiot `compile`, `eval` ja `exec`, joiden avulla voidaan kääntää ja ajaa Python-koodia ohjelman suoritusaikana [Cro12].

Esimerkki suoritusaikaisesta koodin kääntämisestä on listauksessa 7. Esimerkin rivillä 1 luodaan muuttuja, joka sisältää käännettävän koodin merkkijonona. Rivillä 2 käännetään koodi merkkijonosta tavukoodiksi käyttämällä funktiota `compile`, jonka parametrit ovat käännettävä koodi, koodin sisältävän tiedoston nimi ja käännöstila (mode). Nimenä käytetään merkkijonoa `'<string>'` tarkoittamaan, että koodia ei ole luettu tiedostosta. Käännöstila

```
1  def dynamic_c(self):
2    self.a()
3    self.b()
4
5  DynClassC = type(
6    'DynClassC',
7    (MyClassA, MyClassB),
8    {'__doc__':
9     "Dynaaminen moniperinta",
10    'c': dynamic_c
11   })
12
13 olio2 = DynClassC()
14 olio2.c()
15 # foo
16 # bar
```

Listaus 6: Luokan luominen dynaamisesti ajon aikana.

`'single'` kertoo, että käännetään yksi lause (statement). Rivi 4 tulostaa `code_obj` -muuttujan merkkijonoesityksen. Rivillä 7 suoritetaan tavukoodi muuttujasta `code_obj`.

3.5 Tavukoodin tarkastelu ja muokkaus

Standardikirjaston dis-moduulilla voidaan tutkia Python-tavukoodia. Listauksessa 8 on esimerkki tavukoodin tulostamisesta dis-moduulin `dis`-metodilla sekä tulostettu tavukoodi. Esimerkissä käytetty `code_obj` on määritelty listauksessa 7. Tavukoodi on tulostettu Python 2:lla ja se on eri näköinen suoritettaessa esimerkki Python 3:lla.

```
code_str = 'print("Hello world!")'
code_obj = compile(code_str, '<string>', 'single')

print(code_obj)
# <code object <module> ...

exec(code_obj)
# Hello world!
```

Listaus 7: Esimerkki Python-lauseen kääntämisestä tavukoodiksi ohjelman suoritusaikana ja käännetyn koodin ajamisesta [Cro12].

```
import dis

print(dis.dis(code_obj))
#  1           0 LOAD_CONST               0 ('Hello world!')
#              3 PRINT_ITEM
#              4 PRINT_NEWLINE
#              5 LOAD_CONST               1 (None)
#              8 RETURN_VALUE
```

Listaus 8: Python-tavukoodin tarkastelu dis-moduulilla.

3.6 Abstraktin syntaksipuun käsittely

Moduuli parser antaa rajapinnan Python-kääntäjän sisäiseen jäsennyspuuhun ja mahdollistaa sen muokkaamisen. Tämän moduulin lisäämisen jälkeen on kuitenkin standardikirjastoon lisätty ast-moduuli, joka mahdollistaa kääntäjän abstraktin syntaksipuun (abstract syntax tree, AST) luomisen annetun ohjelmakoodin perusteella ja sen muokkaamisen [Klu12]. Funktiolle `compile` voidaan antaa parametrina muokatun AST-puun sisältävä `AST`-luokan olio ja kääntää se tavukoodiksi.

Listauksessa 9 on esimerkki AST-puun luomisesta ja tarkastelusta. Esimerkin rivillä 3 luodaan yksinkertaisesta Python-lauseesta AST-puu `ast.parse`-funktiolla. Rivillä 7 tulostetaan puun rakenne merkkijonona. Riveillä 13 ja 14 ”kuljetaan” läpi puun kaikki solmut ja tulostetaan kunkin solmun luokka.

```
1  import ast
2
3  tree = ast.parse("print('hello world')")
4  exec(compile(tree, filename="<ast>", mode="exec"))
5  # hello world
6
7  print(ast.dump(tree))
8  # Module( body = [Print(
9  #      dest = None,
10 #      values = [Str(s = 'hello world')],
11 #      nl = True)])
12
13 for node in ast.walk(tree):
14   print(type(node))
15 # <class '_ast.Module'>
16 # <class '_ast.Print'>
17 # <class '_ast.Str'>
```

Listaus 9: Abstraktin syntaksipuun tarkastelu ast-moduulilla.

AST-puita voidaan tarkastella myös `ast.NodeVisitor`-luokan avulla. Moduuli tarjoaa apufunktioita puun solmujen tarkasteluun [Klu12].

AST-puun muokkaus onnistuu perimällä `ast.NodeTransformer`-luokka, yliajamalla sen solmuja käsitteleviä metodeita ja kutsumalla luokasta luodun olion `visit`-metodia [Klu12].

3.7 Käännösaikainen metaohjelmointi

Käännettävän ohjelman käännösaikaista metaohjelmointia ei ole suoraan tuettu Pythonissa. Mallien (template) käyttö Python-koodin luomiseen on mahdollista ajamalla mallia käsittelevä ohjelma ja tämän jälkeen kääntää mallin pohjalta luotu koodi. Tähän voidaan käyttää standardikirjastosta löytyvää yksinkertaista, mutta laajennettavaa string-moduulin `Template`-luokkaa tai jotakin Pythonin lukuisista mallimoottoreista (template engine) [Tem].

Käännösaikainen metaohjelmointi on myös lisätty kahteen Pythonista jatkokehitettyyn kieleen, Mythoniin [Rie08] ja Convergeen [Tra05]. Mython-kieltä voidaan kääntää Python-tavukoodiksi, mutta Converge-kielestä käännettyä koodia suoritetaan kielen omalla virtuaalikoneella.

4 Yhteenveto

Python-kielellä suoritusaikainen suoritettavan ohjelman tarkastelu eli reflektio on helppoa. Python-virtuaalikoneen muistissa olevia olioita voidaan tutkia ja muokata suoritusaikana hyvin vapaasti. Laajasta standardikirjastosta on tässä apua.

Uusia luokkia on mahdollista luoda suoritusaikana `type`-funktiolla. Tiedostosta tuotua tai suoritusaikana luotua Python-koodia voidaan kääntää funktiolla `compile` ja käännettyä koodia voidaan ajaa funktiolla `exec`. Abstraktin syntaksipuun muokkaus mahdollistaa tehokkaan tavan tehdä metaohjelmointia suoritusaikana käännettävälle koodille.

Lähteet

[Cro12] Crosta, D.: *Exploring Python Code Objects*, 2012. `http://late.am/post/2012/03/26/exploring-python-code-objects` [17.11.2014].

[Guo14] Guo, P.: *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities*. Communications of The ACM Blog, heinäkuu 2014. `http://cacm.acm.org/blogs/blog-cacm/176450/fulltext` [06.11.2014].

[HB12] Hazzard, K. ja Bock, J.: *Metaprogramming in .NET*. Manning Publications, 2012.

[Ket12] Kettler, R.: *A Guide to Python's Magic Methods*, 2012. `http://www.rafekettler.com/magicmethods.html` [04.12.2014].

[Klu12] Kluyver, T.: *Green Tree Snakes - the missing Python AST docs*, 2012. `http://greentreesnakes.readthedocs.org/en/latest/` [17.12.2014].

[Mar06] Martelli, A.: *Python in a Nutshell*. O'Reilly Media, Inc., 2006.

[Met14] *Metaprogramming – Python 3 Patterns, Recipes and Idioms*, 2014. `http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Metaprogramming.html` [20.11.2014].

[Pil04] Pilgrim, M.: *Dive Into Python*, toukokuu 2004. `http://www.diveintopython.net/` [06.11.2014].

[Pil09] Pilgrim, M.: *Dive Into Python 3*, marraskuu 2009. `http://www.diveintopython3.net/` [10.12.2014].

[Pyt] *The Python Language Reference*. `https://docs.python.org/2/reference/index.html` [06.11.2014].

[Rie08] Riehl, J.: *The Mython Programming Language*, 2008. `http://mython.org/` [16.11.2014].

[Tem] *Templating - Python Wiki*. `https://wiki.python.org/moin/Templating` [16.12.2014].

[Tra05] Tratt, L.: *Compile-time meta-programming in a dynamically typed OO language*. Teoksessa *Proceedings Dynamic Languages Symposium*, sivut 49–64, October 2005.

[VWC13] Van Rossum, G., Warsaw, B. ja Coghlan, N.: *PEP 8 – Style guide for python code*. 2013. `http://www.python.org/dev/peps/pep-0008` [11.11.2014].