

Metaohjelmointi Python-kielellä

Mikko Koho

Seminaarityö

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Helsinki, 16. joulukuuta 2014

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	
Seminaarityö	16. joulukuuta 2014	13	
Tiivistelmä — Referat — Abstract			
<div>Tiivistelmä.</div> <div>ACM Computing Classification System (CCS):</div>			
Avainsanat — Nyckelord — Keywords			
Python, metaohjelmointi			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — Övriga uppgifter — Additional information			

Sisältö

1	Johdanto	1
2	Python-ohjelmointikieli	1
2.1	Syntaksi	2
2.2	Muuttujat	2
2.3	Tietotyytit	3
2.4	Kehitelmät	4
2.5	Luokat ja oliot	5
3	Metaohjelmointi Pythonilla	6
3.1	Reflektio	6
3.2	Dynaamiset luokat	8
3.3	Koodin kääntäminen suoritusaikana	9
3.4	Tavukoodin tarkastelu ja muokkaus	9
3.5	Käännösaikainen metaohjelmointi	11
4	Yhteenveto	11
	Lähteet	12

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 suoritusajasta muistissa olevien olioiden tarkastelua [Pil04].

Tässä seminaarityössä tarkastellaan Python-ohjelmointikielen tarjoamia työkaluja metaohjelmointiin. Alussa käydään läpi Python-kielen perusteita, ja tämän jälkeen tutustutaan metaohjelmointiin Python-kielillä. Metaohjelmoinnista tarkastellaan lähinnä suoritusajasta 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-kielistä 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 sovelluskehikset 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-kielillä toteutettu useissa eri ympäristöissä toimiva suoraan konekielille koodia kääntävä suoritusaikainen (just-in-time) kääntäjä.

CPythonilla käännettyä tavukoodia voidaan ajaa C-kielillä 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 tyyplitetty 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 muuttujan tyyppiä voidaan vaihtaa sijoittamalla siihen eri tyyppinen arvo. Lista 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. Olio tyypin määrittää, mitä metodeja ja ominaisuuksia olio tarjoaa. Osa olioista on muuttumattomia (immutable) ja osa muutettavissa (mutable). Python-kielissä ei ole erikseen vakioita, mutta käytäntönä on käyttää muuttujan nimessä pelkästään isoja kirjaimia, jos arvoa ei ole tarkoitus muuttaa.

Pythonin sisäänrakennettuja tietotyyppejä ovat muun muassa 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 aikatyypin `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 `__getitem__` -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. Funktio `range` palauttaa Python 2:ssa listan ja Python 3:ssa generaattoriolion, jota voidaan käyttää listan tapaan. 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.

```
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([x % y for y in range(2, x-1)])]
13 )
14 # [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Listaus 2: Esimerkki listakehitelmistä ja funktion `range` käytöstä.

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 ”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.

3 Metaohjelmointi Pythonilla

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


```

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    # bar

```

Listaus 3: Esimerkki luokkien moniperinnästä.

3.1 Reflektio

Olion 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 käänösvaiheessa [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 tyyppillinen tapa kuvailla tekstinä ohjelman komponentteja.

Olioita, luokkia ja funktioita voidaan muokata suoritusaikana melko va-

paasti. Esimerkki peruskirjaston `dir` funktion korvaamisesta omalla funktiolla on listauksessa 4. Esimerkin oma funktio palauttaa sille annetut argumentit tekemättä niille mitään. Tämänkaltaisesta kirjastojen ja moduuleiden osien suoritusaikaisesta muokkaamisesta käytetään nimeä ”monkey patching”.

```
1  # -*- coding: utf-8 -*-
2
3  import __builtin__
4
5  def x(*args):
6      return args
7
8  print(dir(__builtin__))
9  # ['ArithmeticError', 'AssertionError', 'AttributeError', ... ]
10
11  __builtin__.dir = x
12
13  print(dir(__builtin__))
14  # (<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`.

Metaluokan metodeilla `__instancecheck__(self, instance)` ja `__subclasscheck__(self, subclass)` voidaan vaikuttaa siihen miten ajoaikainen reflektio käyttäytyy [Ket12, Pyt14]. Näistä ensimmäistä käytetään yliajamaan (override) Pythonin sisäänrakennetun funktion `isinstance` toimintaa, joka kertoo onko instanssi muodostettu tietyistä luokasta. Jälkimmäinen yliajaa sisäänrakennetun funktion `issubclass` toimintaa ja kertoo

```

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.

onko luokka toisen luokan aliluokka.

3.2 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`-metaluokan avulla [Met14]. Kutsumalla `type`:ä kolmella parametrilla saadaan paluuarvona luokka. Parametrit ovat järjestyksessä luokan nimi, perittävät luokat ja sanakirja-tyyppinen muuttuja luokan sisällöstä [Met14]. Esimerkiksi tyhjä luokka voidaan luoda dynaamisesti lausekkeella `C = type('C', (), {})`.

3.3 Koodin kääntäminen suoritusaikana

Pythonissa on sisäänrakennetut funktiot `compile`, `eval` ja `exec`, joiden avulla voidaan kääntää ja ajaa Python-koodia ohjelman suoritusaikana [Cro14].

Esimerkki suoritusaikaisesta koodin kääntämisestä on listauksessa 6. 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 `'single'` kertoo, että käännetään yksi lause (`statement`). Rivi 4 tulostaa `code_obj` -muuttujan merkkijonoesityksen. Rivillä 7 suoritetaan tavukoodi muuttujasta `code_obj`.

```
1 code_str = 'print("Hello world!")'
2 code_obj = compile(code_str, '<string>', 'single')
3
4 print(code_obj)
5 # <code object <module> at 0x7f06341f2cb0, file "<string>", line 1>
6
7 exec(code_obj)
8 # Hello world!
```

Listaus 6: Esimerkki Python-lauseen kääntämisestä tavukoodiksi ohjelman suoritusaikana ja käännetyt koodin ajamisesta [Cro14].

3.4 Tavukoodin tarkastelu ja muokkaus

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

Moduuli `parser` antaa rajapinnan Python-kääntäjän sisäiseen jäsennys-

```

1  # -*- coding: utf-8 -*-
2
3  import dis
4
5  def neliot(iteroitava):
6      '''Palauttaa listan parametrin alkioden neliöistä'''
7      return [x**2 for x in iteroitava]
8
9  dis.dis(neliot)
10 # 4          0 BUILD_LIST          0
11 #           3 LOAD_FAST           0 (iteroitava)
12 #           6 GET_ITER
13 #      >>    7 FOR_ITER           16 (to 26)
14 #           10 STORE_FAST          1 (x)
15 #           13 LOAD_FAST           1 (x)
16 #           16 LOAD_CONST          1 (2)
17 #           19 BINARY_POWER
18 #           20 LIST_APPEND          2
19 #           23 JUMP_ABSOLUTE        7
20 #      >>    26 RETURN_VALUE

```

Lista 7: Python-tavukoodin tarkastelu dis-moduulilla.

puuhun ja mahdollistaa sen muokkaamisen. Tämän moduulin lisäämisen jälkeen on kuitenkin standardikirjastoon lisätty **ast**-moduuli, joka mahdollistaa kääntäjän abstraktin syntaksipuun (abstract syntax tree) luomisen annetun ohjelmakoodin perusteella ja sen muokkaamisen. Funktiolle **compile** voidaan antaa parametrina muokatun AST-puun sisältävä **AST**-luokan olio ja kääntää se tavukoodiksi.

3.5 Käännösaikainen metaohjelmointi

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

Lähteet

- [Cro14] Crosta, D.: *Exploring Python Code Objects*, 2014. <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].
- [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].
- [Pyt14] *Python 2.7.8 documentation*, 2014. <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].