

# Metaohjelmointi Python-kielillä

Mikko Koho

Helsingin Yliopisto

26. marraskuuta 2014

# Metaohjelmointi Python-kielillä

- 1 Johdanto
- 2 Pythonin perusteita
- 3 Reflektio
- 4 Ohjelman muokkaus ajon aikana
- 5 AST-puut

# Metaohjelmointi

- Sellaisen ohjelman tekeminen, joka kirjoittaa uusia ohjelmia
- Ohjelma, joka manipuloi toisia ohjelmia ajon aikana
- Reflektio

# Python

- Dynaaminen olio-ohjelmointikieli
- Dynaamisesti tyypitetty
- Ensimmäinen versio 1991
- Nykyään käytössä versiot 2 ja 3
- Kääntäjiä CPython, Jython, IronPython, PyPy

# Syntaksi

```
lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
joukko = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
monikko = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
print(lista == joukko)
# False
```

```
print(set(lista) == joukko)
# True
```

```
print(lista == list(joukko) == list(monikko))
# True
```

# Syntaksi

```
parents, babies = (1, 1)
while babies < 100:
    print 'This generation has {0} babies'.format(babies)
    parents, babies = (babies, parents + babies)
```

```
# This generation has 1 babies
# This generation has 2 babies
# This generation has 3 babies
# This generation has 5 babies
# This generation has 8 babies
# This generation has 13 babies
# This generation has 21 babies
# This generation has 34 babies
# This generation has 55 babies
# This generation has 89 babies
```

# Luokat

```
class MyClassA(object):  
    def a(self):  
        print('foo')  
  
class MyClassB(object):  
    def b(self):  
        print('bar')  
  
class MyClassC(MyClassA, MyClassB):  
    """Moniperija"""  
    def c(self):  
        self.a()  
        self.b()  
  
olio = MyClassC()  
olio.c()  
  
# foo    # bar
```

# Introspektio

```
for attr in dir(olio):  
    print(attr)
```

```
# __class__  
# __delattr__  
# __dict__  
# __doc__  
# __format__  
# __getattr__  
# __hash__  
# ...  
# a  
# b  
# c
```



# inspect-moduuli

```
import inspect

print(len(dir(inspect)))    # 87

print(inspect.getdoc(inspect.getdoc))
# Get the documentation string for an object.
#
# All tabs are expanded to spaces. To clean up docstrings that are
# indented to line up with blocks of code, any whitespace than can be
# uniformly removed from the second line onwards is removed.

print(inspect.getdoc(inspect))
# Get useful information from live Python objects.    ...

print(inspect.ismodule(inspect))
# True
```

# inspect-moduuli

```
for attr in dir(olio):  
    docstr = str(inspect.getdoc(getattr(olio, attr)))  
    docstr_head = docstr.splitlines()[0]  
    print("olio.%s: %s" % (attr, docstr_head))
```

```
# olio.__class__: Moniperija  
# olio.__delattr__: x.__delattr__('name') <==> del x.name  
# olio.__dict__: dict() -> new empty dictionary  
# olio.__doc__: str(object) -> string  
# olio.__format__: default object formatter  
# olio.__getattr__: x.__getattr__('name') <==> x.name  
# olio.__hash__: x.__hash__() <==> hash(x)  
# olio.__init__: x.__init__(...) initializes x; see help(type(x)) for s  
# ...  
# olio.a: None  
# olio.b: None  
# olio.c: None
```

# inspect-moduuli

```
print(inspect.getsource(olio.c))
#      def c(self):
#          self.a()
#          self.b()

print(inspect.isfunction(olio.c))
# False

print(inspect.ismethod(olio.c))
# True

print(inspect.getmro(MyClassC))
# (<class '__main__.MyClassC'>,
#  <class '__main__.MyClassA'>,
#  <class '__main__.MyClassB'>,
#  <type 'object'>)
```

## Tavukoodin tarkastelu

```
import dis
dis.dis(olio.c)

# 4          0 LOAD_FAST          0 (self)
#           3 LOAD_ATTR          0 (a)
#           6 CALL_FUNCTION      0
#           9 POP_TOP
#
# 5          10 LOAD_FAST        0 (self)
#          13 LOAD_ATTR          1 (b)
#          16 CALL_FUNCTION      0
#          19 POP_TOP
#          20 LOAD_CONST         0 (None)
#          23 RETURN_VALUE
```

# Ohjelman muokkaus ajon aikana

- Olioiden muokkaus sijoittamalla uusia arvoja aiempien tilalle
- Luokan muokkaaminen vaikuttaa siitä luotuihin olioisiin
- Olemassa olevia muuttujia voidaan poistaa `del`-funktiolla
- Myös ulkopuolisia kirjastoja voidaan muokata vapaasti ajon aikana
- Uusien luokkien luominen dynaamisesti

# Dynaamiset luokat

```
def dynamic_c(self):  
    self.a()  
    self.b()
```

```
DynClassC = type(  
    'DynClassC',  
    (MyClassA, MyClassB),  
    {'__doc__':  
        "Dynaaminen moniperinta",  
        'c': dynamic_c  
    })
```

```
olio2 = DynClassC()  
olio2.c()  
# foo  
# bar
```

# Koodin kääntäminen ajon aikana

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

print(code_obj)
# <code object <module> at 0x7f06341f2cb0, file "<string>", line 1>

exec(code_obj)
# Hello world!
```

# AST-puun käsittely

```
import ast

tree = ast.parse("print('hello world')")
exec(compile(tree, filename="<ast>", mode="exec"))
# hello world

print(ast.dump(tree))
# Module( body = [Print(
#     dest = None,
#     values = [Str(s = 'hello world')]],
#     nl = True)])

for node in ast.walk(tree):
    print(type(node))
# <class '_ast.Module'>
# <class '_ast.Print'>
# <class '_ast.Str'>
```



# AST-puun käsittely

```
tree = ast.parse(
    """
    a, b, c = ('foo', 'bar', 'baz')
    print(" - ".join([a, b, c]))
    """)

class StrLister(ast.NodeVisitor):
    def visit_Str(self, node):
        print(node.s)
        self.generic_visit(node)

StrLister().visit(tree)

# foo
# bar
# baz
# -
```

# AST-puun muokkaus

- Onnistuu samaan tapaan kuin NodeVisitor-esimerkissä
- Perittävänä luokkana NodeVisitor:in sijaan NodeTransformer
- Samankaltaiset "visit"-metodit toimivat, paluuarvona annetaan solmu (node)
- ast-moduulista löytyy apumetodeja solmujen muodostamiseen

# Yhteenveto

- Reflektio helppoa
- Laajasta standardikirjastosta apua
- Koodia voi kääntää ja ottaa käyttöön ajon aikana
- AST-puun muokkaus