

Programavimo kalba **Python**

aštuntoji paskaita

Marius Gedminas
<mgedmin@b4net.lt>

<http://mg.b4net.lt/python/>





Dekinatoriai ir metaklasės



gili magija



Klasės ir metodai (pakartojimas)

```
class ManoKlase(object):  
    def metodas(self, x):  
        print x  
  
    def statinism(x):  
        print x  
    statinism = staticmethod(statinism)  
  
    def klasesm(cls, x):  
        print x  
    klasesm = classmethod(klasesm)
```

```
> > > obj = ManoKlase()
```

```
> > > obj.metodas(42)
```

```
42
```

```
> > > ManoKlase.statinism(42)
```

```
42
```

```
> > > ManoKlase.klasesm(42)
```

```
42
```

Kas per paukštis yra klasės metodai
ir su kuo jie valgomi?

```
class Figura(object):
```

```
    def __init__(self, x, y): ...
```

```
    def atsitiktine(cls):
```

```
        return cls(random.gauss(0, 100),  
                  random.gauss(0, 100))
```

```
    atsitiktine = classmethod(atsitiktine)
```

```
class Varliukas(Figura): ...
```

```
class Zuvyte(Figura): ...
```




Grižtam prie sintaksės

```
def atsitiktine (cls):  
    return cls(random.gauss(0, 100),  
               random.gauss(0, 100))  
atsitiktine = classmethod(atsitiktine)
```



Tris kartus kartoti tą patį vardą
nepatogu



Python 2.4 įvedė naują sintaxę -- dekoratorius

```
@classmethod  
def atsitiktine (cls):  
    return cls(random.gauss(0, 100),  
               random.gauss(0, 100))
```

=

```
def atsitiktine (cls):  
    return cls(random.gauss(0, 100),  
               random.gauss(0, 100))  
atsitiktine = classmethod(atsitiktine)
```



Nieko per daug stebuklingo



Bet galima daryti triukus

```
class Rect(object):
```

```
    ...
```

```
    def _calc_width(self):
```

```
        return self.x2 - self.x1
```

```
    width = property(_calc_width)
```

```
>>> r = Rect(10, 10, 630, 400)
```

```
>>> r.width
```

```
620
```

```
>>> r.x2 = 400; r.width
```

```
390
```



```
class Rect(object):
```

```
...
```

```
    @property
```

```
    def width(self):
```

```
        return self.x2 - self.x1
```

```
>>> r = Rect(10, 10, 630, 400)
```

```
>>> r.width
```

```
620
```

```
>>> r.x2 = 400; r.width
```

```
390
```

```
class Rect(object):
```

```
...
```

```
def _calc_width(self): ...
```

```
def _set_width(self, width):
```

```
    self.x2 = self.x1 + width
```

```
width = property(_calc_width,  
                 _set_width)
```

```
>>> r = Rect(10, 10, 630, 400)
```

```
>>> r.width = 20; r.x2
```

```
30
```

```
class Rect(object):  
    ...  
    @apply  
    def width():  
        def get(self):  
            return self.x2 - self.x1  
        def set(self, width):  
            self.x2 = self.x1 + width  
    return property(get, set)
```

```
# google for rwproperty.py
class Rect(object):
    ...
    @rwproperty.getproperty
    def width(self):
        return self.x2 - self.x1
    @rwproperty.setproperty
    def width(self, new_value):
        self.x2 = self.x1 + new_value

>>> r = Rect(10, 10, 630, 400)
>>> r.width
620
>>> r.x2 = 400; r.width
390
```

Aukštasis pilotažas: rašome savo dekoratorių

```
def debug(fn):  
    def wrapped_fn(*args, **kw):  
        print "Entering %s" % fn.__name__  
        return fn(*args, **kw)  
    wrapped_fn.__name__ = fn.__name__  
    return wrapped_fn
```

```
@debug  
def compute(x, y):  
    return x + y
```

```
>>> compute(2, 3)  
Entering compute  
5
```

Tikrame kode tokių triukų geriau
nenaudoti

Kodo įskaitomumas svarbiau!

Bet debuginant praverčia, nes
pakanka pridėti vieną eilutę
prieš bet kokią funkciją ar metodą

```
def breakpoint(fn):  
    def wrapped_fn(*args, **kw):  
        import pdb; pdb.set_trace()  
        return fn(*args, **kw)  
    wrapped_fn.__name__ = fn.__name__  
    return wrapped_fn
```

```
@breakpoint  
def compute(x, y):  
    return x + y
```

```
>>> compute(2, 3)  
> <stdin>(4)wrapped_fn()  
(pdb)
```

```
def timed(fn):  
    def wrapped_fn(*args, **kw):  
        start = time.time()  
        try:  
            return fn(*args, **kw)  
        finally:  
            print "%s took %.3f seconds" % (  
                fn.__name__,  
                time.time() - start)  
    wrapped_fn.__name__ = fn.__name__  
    return wrapped_fn
```

```
@timed
def compute(n):
    for i in range(n):
        time.sleep(0.2)

>>> compute(10)
compute took 1.999 seconds
```



Dekinatorius profiavimui

<http://mg.pov.it/blog/profiling.html>

```
from profilehooks import profile
```

```
@profile
```

```
def compute(n):  
    for i in range(n):  
        time.sleep(0.2)
```

```
>>> compute(2, 3)
```

```
>>> ^D
```

```
*** PROFILER RESULTS ***
```

```
compute (<stdin>:1)
```

```
function called 1 times
```

```
1 function calls in 0.000 CPU seconds
```

```
Ordered by: internal time, call count
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	<stdin>:1(compute)
0	0.000		0.000		profile:0(profiler)



Dekordinatorius padengimo analizei

<http://mg.pov.lt/blog/profiling.html>


```
from profilehooks import coverage
```

```
@coverage  
def compute(n):  
    if n % 2 == 1:  
        print "a"  
    else:  
        print "b"
```

```
$ python x.py  
b
```

```
*** COVERAGE RESULTS ***  
compute (x.py:3)  
function called 1 times
```

```
    @coverage  
    def compute(n):  
1:         if n % 2 == 1:  
>>>>>>         print "a"  
                else:  
1:         print "b"
```

```
1 lines were not executed.
```



Dekinatoriai su parametrais

```
def debug(prefix):  
    def decorator(fn):  
        def w_fn(*args, **kw):  
            print "%s %s" % (prefix,  
                             fn.__name__)  
            return fn(*args, **kw)  
        w_fn.__name__ = fn.__name__  
        return w_fn  
    return decorator
```

```
@debug('==>')  
def compute(x, y):  
    return x + y
```

```
>>> compute(2, 3)  
==> compute  
7
```



(Pailsinkim smegenis)



Metaklasės

Kas yra objekto klasė?


```
>>> x = 5; x.__class__  
<type 'int'>  
>>> y = 'abc'; y.__class__  
<type 'str'>
```



Klasė taip pat yra objektas

Kas yra objekto klasė,
kai pats objektas yra klasė?

```
>>> x = 5; x.__class__  
<type 'int'>  
>>> x.__class__.__class__  
<type 'type'>
```

Tai yra paprasta klasė.
Bet kadangi ji yra klasės klasė,
ji vadinama metaklase.

5 yra objektas

(int klasės egzempliorius)

int yra objektas

(type klasės egzempliorius)

type yra objektas

(type klasės egzempliorius)



Mazgelis užrištas

Turėdami klasę galime kurti jos
egzempliorių
kviesdami klasę kaip funkciją


```
>>> int
<type 'int'>
>>> int(5)
5
```

Metaklasė irgi klasė

Jos egzemplioriai -- klasės

```
class NaujaKlase(object):
```

```
    a = 1
```

```
    def b(self):
```

```
        return 2 + self.a
```

=

```
def b(self):
```

```
    return 2 + self.a
```

```
NaujaKlase = type('NaujaKlase', (), {'a': 1, 'b': b})
```

```
del b
```

Yra daugiau nei viena metaklasė
bet dabar į tai nesigilinsime