

# Napredne tehnike programiranja u Python-u

Prof. dr Igor Dejanović ([igord@uns.ac.rs](mailto:igord@uns.ac.rs))

Kreirano 2025-11-17 Mon 00:01, pritisni ESC za mapu, Ctrl+Shift+F za pretragu, ? za pomoć

# Sadržaj

1. Specijalne metode
2. *Properties*
3. *List comprehensions* i generatori
4. Deskriptori
5. Dekoratori(*Decorators*)
6. `functools` modul - podrška za funkcije višeg reda.



# Specijalne metode

## Specijalne metode

- Često se zovu i **magične** metode
- Posebno se tretiraju od strane Python interpretera tj. imaju posebnu semantiku
- Format naziva je `__xxx__`
- Neki od primera:
  - `__init__`
  - `__str__`
  - `__eq__`
  - `...`
- Implementacija protokola

## Iterabilni objekti

- Moguće ih je koristiti u npr. `for` petlji.

```
for i in iterabilni_objekat:
```

- Poziv ugrađene funkcije `iter` nad iterabilnim objektom vraća iterator objekat.
- Kada je potrebna iteracija Python poziva `__iter__` metodu nad našim objektom.  
Ova funkcija treba da vrati iterator objekat.
- Iterator objekti implementiraju tzv. `iterator` protokol.
  - `__next__` metoda – sledeći element ili izuzetak `StopIteration` ukoliko smo stigli do kraja.

## Iterabilni objekti - primer

- Primer `TextXMetaModel` klasa `textX` projekta.

```
def __iter__(self):
    """
    Iterate over all classes in the current namespace and imported
    namespaces.
    """

    # Current namespace
    for name in self._current_namespace:
        yield self._current_namespace[name]

    # Imported namespaces
    for namespace in \
            self._imported_namespaces[self._namespace_stack[-1]]:
        for name in namespace:
            # yield class
            yield namespace[name]
```

- U ovom slučaju `__iter__` metoda je generator (vraća elemente sa `yield`).

## Provera pripadnosti kolekciji

- Operator `in` služi za testiranje pripadnosti:

```
if a in neka_kolekcija:
```

- Ukoliko želimo da omogućimo `in` test sa našim objektima potrebno je definisati specijalnu metodu `__contains__`:

```
def __contains__(self, name):  
    """  
    Check if given name is contained in the current namespace.  
    The name can be fully qualified.  
    """  
    try:  
        self[name]  
        return True  
    except KeyError:  
        return False
```

## Pristup elementu kolekcije po ključu

- Specijalna metoda `__getitem__` omogućava upotrebu operatora `[]`.

```
>>> a = [1, 2, 3]
>>> a.__getitem__(2)
3
>>> b = { "foo": 45, "bar": 34}
>>> b.__getitem__(34)
-----
KeyError                                     Traceback (most recent call last)
...
KeyError: 34
>>> b.__getitem__("foo")
45
```

## Pristup elementu kolekcije po ključu - primer

```
def __getitem__(self, name):
    Search for and return class and peg_rule with the given name.
    Returns:
        TextXClass, ParsingExpression
    """
    if "." in name:
        # Name is fully qualified
        namespace, name = name.rsplit('.', 1)
        return self.namespaces[namespace][name]
    else:
        # If not fully qualified search in the current namespace
        # and after that in the imported_namespaces
        if name in self._current_namespace:
            return self._current_namespace[name]

        for namespace in \
            self._imported_namespaces[self._namespace_stack[-1]]:
            if name in namespace:
                return namespace[name]

    raise KeyError("{} metaclass does not exists in the metamodel "
                  .format(name))
```

## Pristup atributu objekta

- Dve specijalne metode: `__getattr__` i `__getattribute__`

```
obj.neki_atribut
```

## \_\_getattr\_\_

- Poziva se ukoliko standardnim mehanizmima nije pronađen atribut objekta. Kao parametar prima ime atributa i vraća njegovu vrednost ukoliko postoji ili podiže izuzetak `AttributeError` ukoliko atribut ne postoji.

```
class A:  
    def __init__(self):  
        self.additional = {'foo': 5, 'bar': 7.4}  
        # .a će biti pronađeno standardnim mehanizmom  
        self.a = 3  
  
    def __getattr__(self, key):  
        if key in self.additional:  
            return self.additional[key]  
        else:  
            raise AttributeError  
  
    if __name__ == '__main__':  
        a = A()  
        print(a.a)  
        print(a.foo)  
        print(a.bla)
```

## Postavljanje vrednosti i brisanje atributa

- `object.__setattr__(self, name, value)` - kada se pozove `object.name = value`
  - Obratiti pažnju na rekurziju! Ne raditi u telu metode `self.name = value` već `self.__dict__[name] = value`.
- `object.__delattr__(self, name)` - kada se pozove `del object.name`

## `__getattribute__`

- Specijalna metoda nižeg nivoa.
- Podrazumevana implementacija obavlja sledeću pretragu:
  - Prvo se proverava `__dict__` rečnik instance a zatim klasa prateći MRO lanac.
  - Ukoliko se atribut ne pronađe poziva se `__getattr__`
- Ovu metodu je retko potrebno redefinisati.
- Paziti prilikom pisanja na beskonačanu rekurziju! Najbolje je pozvati na kraju nadimplementaciju `object.__getattribute__(self, name)`

## Operatori

- Svi operatori u Python-u su definisani specijalnim metodama. Na primer:

- `-` - `__sub__`
- `+` - `__add__`
- `*` - `__mul__`
- `+=` - `__iadd__` - (za **mutable** objekte)
- ...

## Poređenje objekata

- Ukoliko želimo da instance naše klase mogu da se porede (npr. da bi mogli da sortiramo niz objekata) potrebno je implementirati specijalne metode:
  - `__eq__` za operator jednakosti `==`
  - `__ne__` za operator nejednakosti `!=`
  - `__lt__` za operator **manje** - `<`
  - `__le__` za operator **manje ili jednako** - `<=`
  - `__gt__` za operator **veće** - `>`
  - `__ge__` za operator **veće ili jednako** - `>=`
- Nije potrebno ručno definisati sve operatore jer su prirodno međuzavisni. Python u sklopu modula `functools` nudi dekorator `total_ordering` (videti u sekciji `functools`) koji automatski kreira nedostajuće operatore poređenja.



## *Properties*

## Properties

- Tzv. kalkulisani ili izvedeni atributi.
- Generalizacija *getter* i *setter* mehanizma.
- Sintaksa ostaje kao kod direktnog pristupa atributu.

## Properties - primer

```
class ...:  
    ...  
    @property  
    def full_file_name(self):  
        return os.path.join(self.file_path, self.file_name)  
  
    @full_file_name.setter  
    def full_file_name(self, filename):  
        path_name, file_name = os.path.split(filename)  
        self.file_path = path_name  
        self.file_name = file_name  
        model_name, __ = os.path.splitext(file_name)  
        self.name = model_name
```

*Property* atributima se pristupa kao običnim atributima:

```
obj.full_file_name = '/neka/putanja/neko_ime.ext'
```

- Objekat čuva attribute `file_path` i `file_name`.
- Atribut `full_file_name` je izведен.

## `__setattr__` i `@property`

```
class A:  
    @property  
    def a(self):  
        print("get")  
        return 5  
    @a.setter  
    def a(self, val):  
        print("set")  
  
    def __setattr__(self, name, value):  
        if name == 'a':  
            print('setattr')  
        super().__setattr__(name, value)  
  
a = A()  
a.a = 10  
out = a.a
```

```
setattr  
set  
get
```



## *List comprehensions i generatori*

## List comprehensions

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    squares.append(n * n)

# Ekvivalentno
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

# Opšti oblik sintakse  
[expression for item1 in iterable1 if  
condition1  
 for item2 in iterable2 if  
condition2  
 ...  
 for itemN in iterableN if  
conditionN ]

# Što je ekvivalentno sa  
s = []
for item1 in iterable1:  
 if condition1:  
 for item2 in iterable2:  
 if condition2:  
 ...  
 for itemN in iterableN:  
 if conditionN: s.append(expression)

## List comprehensions primeri

```
a = [-3, 5, 2, -10, 7, 8]
b = ['abc', 'abc']

c = [2*s for s in a]
d = [s for s in a if s >= 0]
e = [(x, y) for x in a
      for y in b
      if x > 0]

f = [(1,2), (3,4), (5,6)]
g = [math.sqrt(x*x + y*y)
      for x, y in f]
```

```
# c = [-6, 10, 4, -20, 14, 16]
# d = [5, 2, 7, 8]
# e = [(5, 'a'), (5, 'b'), (5, 'c'),
#       (2, 'a'), (2, 'b'), (2, 'c'),
#       (7, 'a'), (7, 'b'), (7, 'c'),
#       (8, 'a'), (8, 'b'), (8, 'c')]
#
# g = [2.23606, 5.0, 7.81024]
```

## Generator izrazi

Slično kao *list comprehensions* ali ne kreiraju listu već generator objekat koji izračunava vrednosti na zahtev (lenja evaluacija).

```
# Opšti oblik sintakse
(expression for item1 in iterable1 if
condition1
    for item2 in iterable2 if
condition2
        ...
            for itemN in iterableN if
conditionN )
```

```
>>> a = [1, 2, 3, 4]
>>> b = (10*i for i in a)
>>> b
<generator object at 0x590a8>
>>> b.next()
10
>>> b.next()
20
...
...
```

## Generator izrazi - primer

```
f = open("data.txt")
lines = (t.strip() for t in f)

comments = (t for t in lines if t[0] == '#')
for c in comments:
    print(c)

# Uvek se može konvertovati u listu
clist = list(comments)
```



# Deskriptori

## Descriptors

- Generalizacija prilagođavanja pristupu atributima objekata.
- Npr. *properties* iz prethodne sekcije su implementirani mehanizmom deskriptora.
- Ukoliko interpreter pronađe atribut na nivou klase i taj atribut je objekat koji ima neku od metoda `__get__`, `__set__`, `__del__` tada će ovim metodama biti prosleđeno dobavljanje, postavljanje ili brisanje atributa respektivno.

## Problem sa *properties*

```
class Order:  
    def __init__(self, name, price, quantity):  
        self._name = name  
        self.price = price  
        self._quantity = quantity  
  
    @property  
    def quantity(self):  
        return self._quantity  
  
    @quantity.setter  
    def quantity(self, value):  
        if value < 0:  
            raise ValueError('Cannot be negative.')  
        self._quantity = value  
  
    def total(self):  
        return self.price * self.quantity  
  
apple_order = Order('apple', 1, 10)  
print(apple_order.total())  
try:  
    apple_order.quantity = -10  
except ValueError as e:  
    print('Woops:', str(e))
```

10

Woops: Cannot be negative.

<https://dev.to/dawranliou/writing-descriptors-in-python-36>

## Rešenje upotrebom deskriptora (1)

```
class NonNegative:

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Cannot be negative.')
        instance.__dict__[self.name] = value

    def __set_name__(self, owner, name):
        self.name = name
```

- <https://dev.to/dawranliou/writing-descriptors-in-python-36>
- <https://docs.python.org/3/howto/descriptor.html>

## Rešenje upotrebom deskriptora (2)

```
class Order:  
    price = NonNegative()  
    quantity = NonNegative()  
  
    def __init__(self, name, price, quantity):  
        self.name = name  
        self.price = price  
        self.quantity = quantity  
  
    def total(self):  
        return self.price * self.quantity  
  
apple_order = Order('apple', 1, 10)  
print(apple_order.total())  
  
try:  
    apple_order.quantity = -10  
except ValueError as e:  
    print('Woops:', str(e))
```

```
10  
Woops: Cannot be negative.
```

<https://dev.to/dawranliou/writing-descriptors-in-python-36>



## Dekoratori(*Decorators*)

## Dekoratori

- Dekorator obrazac.
- Funkcije koje prihvataju kao parametar funkciju (ili uopšte `callable`) i vraćaju izmenjenu verziju.

```
@trace
def square(x):
    return x*x

# Ovo je ekvivalentno sa
def square(x):
    return x*x
square = trace(square)
```

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log", "w")
def trace(func):
    if enable_tracing:
        def callf(*args, **kwargs):
            debug_log.write("Calling %s: %s, %s\n" %
                           (func.__name__, args, kwargs))
            r = func(*args, **kwargs)
            debug_log.write("%s returned %s\n" %
                           (func.__name__, r))
            return r
    return callf
```

```
else:  
    return func
```

## Dekoratori (2)

Mogu da se stekuju.

```
@foo  
@bar  
@spam  
def grok(x):  
    pass
```

je isto što i

```
def grok(x):  
    pass  
grok = foo(bar(spam(grok)))
```

## Dekoratori (3)

Mogu da imaju parametre.

```
@eventhandler('BUTTON')
def handle_button(msg):
    ...

@eventhandler('RESET')
def handle_reset(msg):
    ...

# Sto je ekvivalentno sa
def handle_button(msg):
    ...
temp = eventhandler('BUTTON')
handle_button = temp(handle_button)
```

```
# Event handler decorator
event_handlers = { }
def eventhandler(event):
    def register_function(f):
        event_handlers[event] = f
        return f
    return register_function
```





`functools` modul - podrška za funkcije višeg reda.

## partial

- Delimična (parcijalna) primena funkcije.
- Određeni parametri se **zamrzavaju**. Nova funkcija prihvata manji broj parametara.

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Konverzija stringa broja u bazi 2 u int.'
>>> basetwo('10010')
18

>>> stepen = lambda a, b: a ** b
>>> kvadrat = partial(stepen, b=2)
>>> kvadrat(5)
25
>>> kub = partial(stepen, b=3)
>>> kub(5)
125

>>> from operator import mul
>>> dvaputa = partial(mul, 2)
>>> dvaputa(10)
20
>>> triputa = partial(mul, 3)
>>> triputa(5)
15
```



## reduce

- Primena date funkcije koja prima dva parametra na iterabilnu kolekciju s leva na desno tako što se kao prvi parametar koristi rezultat prethodne evaluacije dok se kao drugi parametar koristi sledeći element kolekcije.

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

## izračunava

```
((1+2)+3)+4)+5)
```

- Postoji i kao ugrađena (**build-in**) funkcija u Python 2.

## reduce ekvivalentan Python kod

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            initializer = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    accum_value = initializer
    for x in it:
        accum_value = function(accum_value, x)
    return accum_value
```

## Primer: množenje niza elemenata

```
niz = range(1, 10)
```

**for** petlja:

```
proizvod = 1
for elem in niz:
    proizvod *= elem
```

**reduce**:

```
proizvod = reduce(lambda x, y: x*y, niz)
```

## Kreiranje funkcije za množenje niza elemenata

Kompozicija funkcija `partial` + `reduce` (funkcionalno):

```
from functools import reduce, partial
amul = partial(reduce, lambda x, y: x*y)
# ili upotrebom mul operatora
from operator import mul
amul = partial(reduce, mul)
```

Sa `for` petljom (imperativno):

```
def amul(niz):
    proizvod = 1
    for elem in niz:
        proizvod *= elem
    return proizvod
```

Oba primera kreiraju funkciju `amul` koja množi elemente prosleđenog iterabilnog objekta:

```
amul(range(1, 100))
```



## update\_wrapper i wraps

- Kod dekoracije funkcija ažurira dekorisanu funkciju da spolja "izgleda" kao originalna.

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print 'Calling decorated function'
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print 'Called example function'
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

## total\_ordering

- “Dopuna” specijalnih metoda za poređenje. Koristi se kao dekorator klase.
- Klasa treba da definiše jednu od `__lt__`, `__le__`, `__gt__`, ili `__ge__()` metoda uz `__eq__`.

```
@total_ordering
class Student:
    def __is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self.__is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self.__is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

<https://docs.python.org/3/library/functools.html>

## *Least recently used (LRU) keš*

Keširanje povratne vrednosti funkcije u cilju optimizacije sporijih funkcija.

```
@lru_cache(maxsize=32)
def get_pep(num):
    '''Retrieve text of a Python Enhancement Proposal'''
    resource = 'https://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

[https://en.wikipedia.org/wiki/Cache\\_replacement\\_policies#Least\\_recently\\_used\\_\(LRU\)](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU))

Speaker notes