PROGRAMIRANJE U INTEGRISANIM TEHNOLOGIJAMA

Oznaka predmeta: PIT

Predavanje broj: 08

Nastavna jedinica: Metaprogramiranje. Heuristično programiranje.

Nastavne teme:

Dekoratori funkcija. Dekorator relizovan kao klasa. Parametrizovani dekorator. Metaklase. Heuristično programiranje.

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

David Beazley and Brian K. Jones: "Python cookbook", O'REILLY, 2013.

Closure

- Zatvarač (closure) : ideja je da se "uhvate" lokalne promenljive, u datom primeru to su stvarni argumenti prosleđeni funkciji make_adder

```
def make_adder(x, y):
    def add():
        return x + y
    return add

a = make_adder(2, 3)
b = make_adder(10, 20)
print(a())
print(b())

# <function make_adder.<locals>.add at 0x00000175D8467598>
# <function make_adder.<locals>.add at 0x00000175D8467620>
# 5
# 30
```

Primetiti šta vraća funkcija make_adder i šta je sa a i b nakon poziva funkcije make_adder.

Metaprogramiranje: dekoratori

- Metaprogramiranjem u opštem slučaju programi postaju podaci.
- Ideja je kreiranje funkcija i klasa čiji je glavni cilj manipulacija kodom (izmena koda, generisanje koda ili umetanje postojećeg koda).
- Razmatraju se : dekoratori funkcija , dekoratori kao klase i metaklase.
- Dekorator je funkcija koja kreira wrapper oko druge funkcije
- Postavljanje wrapper-a funkcije:

```
import time
from functools import wraps

def timethis(func):
    'ovaj decorator --> pokazuje vreme izvrsavanja pozvane funkcije'
    @wraps(func) #ocuvanje metapodataka funkcije
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)  #svi argumenti
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

Metaprogramiranje: dekoratori

```
'''Koriscenje'''
@timethis
def countdown(n):
    'Counts down'
    while n > 0:
        n -= 1
    print('ha')

# ovaj blok je isto kao
# def countdown(n):
# ...
# countdown = timethis(countdown)

countdown( 100000)
countdown(1000000)

# ha
# countdown 0.01562047004699707
# ha
# countdown 0.09374427795410156
```

Metaprogramiranje: dekoratori

- Ako se ne bi koristio decorator @wraps, tada bi se desilo sledeće:

npr. neka je prethodna funkcija countdown malo "izmenjena":

```
@timethis
def countdown(n:int):
    'Counts down'
    while n > 0: n -= 1
    print('ha')

countdown(100000)
# ha
# countdown 0.01562213897705078

print(countdown.__name__)
print(countdown.__doc__)
print(countdown.__annotations__)
# countdown
# Counts down                → bez @wraps ne vraca nista
# {'n': <class 'int'>}       → bez @wraps vraca {}
# direktan poziv wrap-ovane funkcije
print(countdown.__wrapped__(100000))
# ha
# None
```

Metaprogramiranje: __wrapped__

- Aktiviranje samo wrap-ovane funkcije:

```
orig_countdown = countdown.__wrapped__
orig_countdown(5)
# ha
```

- U slučaju više dekoratora dodeljenih funkciji bolje je izbegavati `__wrapped__` (može se desiti *bug*).

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper
```

Metaprogramiranje: __wrapped__

```
@decorator1
@decorator2
def add(x, y):
    print(x + y)
add(2, 3)
# Decorator 1
# Decorator 2
# 5
add.__wrapped__(2, 3)
# 5
```

- Moguće je parametrizovati dekorator

```
from functools import wraps
import logging
logging.basicConfig(level=logging.DEBUG)
# logging.basicConfig(  filename="test.log",
#                       level=logging.DEBUG,
#                       format="%(asctime)s:%(levelname)s:%(message)s")
def logged(level, name=None, message=None):
    '''
    Add logging to a function.
      level   -  logging level:CRITICAL,ERROR,WARNING,INFO,DEBUG
      name    -  logger name,
      message -  log message. '''
```

Metaprogramiranje: parametrizacija dekoratora

```
    def decorate(func):
        # print('-decorate-')
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            # print('-wrapper-')
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

@logged(logging.DEBUG) # kao add = logged(logging.DEBUG)(add)
def add(x, y): return x + y

@logged(logging.CRITICAL, 'example')
def spam(): print('SPAM')

add(2,3) # DEBUG:__main__:add
spam()   # CRITICAL:example:spam
         # SPAM → nije odredjeno mesto ispisa
```

Dekorator sa parametrima i funkcija sa parametrima

```
def decorator_function_with_arguments(arg1, arg2, arg3):
    def wrap(f):
        print("Inside wrap()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:", arg1, arg2, arg3)
            f(*args)
            print("After f(*args)")

        return wrapped_f
    return wrap

@decorator_function_with_arguments("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print('sayHello arguments:', a1, a2, a3, a4)

# Inside wrap()
print("After decoration")
# After decoration
print("Preparing to call sayHello()")
# Preparing to call sayHello()
```

Dekorator sa parametrima i funkcija sa parametrima

```
sayHello("say", "hello", "argument", "list")
   # Inside wrapped_f()
   # Decorator arguments: hello world 42
   # sayHello arguments: say hello argument list
   # After f(*args)

print("after first sayHello() call")
   # after first sayHello() call

sayHello("a", "different", "set of", "arguments")
   # Inside wrapped_f()
   # Decorator arguments: hello world 42
   # sayHello arguments: a different set of arguments
   # After f(*args)

print("after second sayHello() call")
   # after second sayHello() call
```

Metaprogramiranje: parametrizacija dekoratora

- Korisničko podešavanje atributa za programsko kontrolisanje ponašanja dekoratora u vreme izvršavanja.

```
from functools import wraps, partial
import logging

# decorator – objektu dodaje atribut koji je funkcija
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    ''' Kao ranije'''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
```

Metaprogramiranje: parametrizacija dekoratora

```
        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper
    return decorate

@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

Metaprogramiranje: parametrizacija dekoratora

```
logging.basicConfig(level=logging.DEBUG)
add(2, 3)
# DEBUG:__main__:add

# Change the log message
add.set_message('Add called')
add(2, 3)
# DEBUG:__main__:Add called

# Change the log level
add.set_level(logging.WARNING)
add(2, 3)
# WARNING:__main__:Add called
```

- Drugi način da se uradi korisničko podešavanje atributa za programsko kontrolisanje ponašanja dekoratora u vreme izvršavanja je kao što sledi:

```
from functools import wraps, partial
import logging
def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)
```

Metaprogramiranje: parametrizacija dekoratora

```
    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__

    @wraps(func)
    def wrapper(*args, **kwargs):
        log.log(level, logmsg)
        return func(*args, **kwargs)

    return wrapper

@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')

logging.basicConfig(level=logging.DEBUG)
add(1,2)
spam()
```

Dekorator: metoda, klasna metoda

- Dekorator kao metoda objekta i dekorator kao klasna metoda.
- Ideja je da se metoda objekta i metoda klase upotrebe kao dekoratori.

```
from functools import wraps

class A:
    # decorator kao metoda objekta
    def decorator1(self, func):
        print('a')
        @wraps(func)
        def wrapper1(*args, **kwargs):
            print('decorator 1')
            return func(*args, **kwargs)

        return wrapper1

    # decorator kao klasna metoda
    @classmethod
    def decorator2(cls, func):
        print('b')
```

Dekorator: metoda, klasna metoda

```
    @wraps(func)
    def wrapper2(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)

    return wrapper2

a = A()
@a.decorator1
def spam(): print('SPAM')

@A.decorator2
def grok(): print('GROK')

spam()
grok()
# a
# b
# decorator 1
# SPAM
# Decorator 2
# GROK
```

Dekoratorska klasa

- Klasa kao dekorator.

```
class my_decorator(object):

    def __init__(self, f):
        print("inside my_decorator.__init__()")
        f() # samo dokaz da je f kompletirana

    def __call__(self):
        print("inside my_decorator.__call__()")

@my_decorator
def aFunction():
    print("inside aFunction()")

print("Finished decorating aFunction()")

aFunction()

# inside my_decorator.__init__()
# inside aFunction()
# Finished decorating aFunction()
# inside my_decorator.__call__()
```

Dekoratorska klasa

```
class entry_exit(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)
@entry_exit
def func1():
    print("inside func1()")
@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
# Entering func1
# inside func1()
# Exited func1
# Entering func2
# inside func2()
# Exited func2
```

Dekoratorska klasa : dekorator bez argumenata

```
class decorator_without_arguments(object):
    def __init__(self, f):
        print("Inside __init__()")
        self.f = f

    def __call__(self, *args):
        print("Inside __call__()")
        self.f(*args)
        print("After self.f(*args)")

@decorator_without_arguments
def sayHello(a1,a2,a3,a4):
    print('sayHello arguments:', a1, a2, a3, a4)
# Inside __init__()

print("After decoration")
# After decoration
print("Preparing to call sayHello()")
# Preparing to call sayHello()
```

Dekoratorska klasa : dekorator bez argumenata

```
sayHello("say", "hello", "argument", "list")
# Inside __call__()
# sayHello arguments: say hello argument list
# After self.f(*args)

print("After first sayHello() call") # After first sayHello() call

sayHello("a", "different", "set of", "arguments")
# Inside __call__()
# sayHello arguments: a different set of arguments
# After self.f(*args)

print("After second sayHello() call") # After second sayHello() call
```

Dekoratorska klasa : dekorator sa argumentima

```
class decorator_with_arguments(object):
    def __init__(self, arg1, arg2, arg3):
        print("Inside __init__()")
        self.arg1 = arg1
        self.arg2 = arg2;
        self.arg3 = arg3

    def __call__(self, f):
        print("Inside __call__()")
        def wrapped_f(*args):
            print("Inside wrapped_f()")
            print("Decorator arguments:",self.arg1,self.arg2,self.arg3)
            f(*args)
            print("After f(*args)")
        return wrapped_f

@decorator_with_arguments("hello", "world", 42)
def sayHello(a1,a2,a3,a4):
    print('sayHello arguments:', a1, a2, a3, a4)
# Inside __init__()
# Inside __call__()
print("After decoration") # After decoration
```

Dekoratorska klasa : dekorator sa argumentima

```
print("Preparing to call sayHello()")
# Preparing to call sayHello()

sayHello("say", "hello", "argument", "list")
# Inside wrapped_f()
# Decorator arguments: hello world 42
# sayHello arguments: say hello argument list
# After f(*args)
print("after first sayHello() call")
# after first sayHello() call

sayHello("a", "different", "set of", "arguments")
# Inside wrapped_f()
# Decorator arguments: hello world 42
# sayHello arguments: a different set of arguments
# After f(*args)
print("after second sayHello() call")
# after second sayHello() call
```

Metaprogramiranje

- Metaklase kreiraju klase, dok klase kreiraju objekte.
  - Klasa se ponaša kao "objekat".

```
class Foo: pass

Foo.field = 42
x = Foo()
print(x.field) # 42

Foo.field2 = 99
print(x.field2) # 99

Foo.method = lambda self: "Hi!"
print(x.method()) # 'Hi!'
```

- Ekvivalenti:

```
class C: pass
C = type('C',(),{})
```

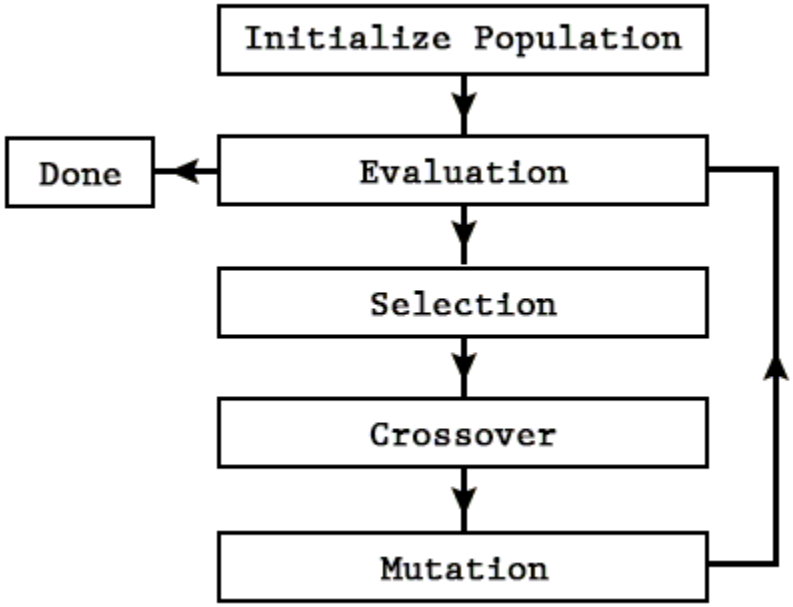
U skladu sa prethodnim:

```
def howdy(self, you):
    print("Howdy, " + you)

MyList = type('MyList', (list,), dict(x=42, howdy=howdy))
```

Heurističko programiranje

```
ml = MyList()
ml.append("Camembert")
print(ml)                    # ['Camembert']
print(ml.x)                  # 42
ml.howdy("John")             # Howdy, John
print(ml.__class__.__class__) # <class 'type'>
```

- U okviru heurističkog programiranja sledi program jednostavnog GA (genetski algoritam).
- Ideja je sledeća:



Jednostavni GA

- Neka je cilj da se od inicijalne populacije niza karaktera kroz generacije evoluira najbolje rešenje čiji je idealni cilj niz "`Hello, world`".
- Inicijalna populacija od `POP_SIZE` hromozoma se kreira generisanjem slučajnih nizova karaktera (vidljivih na konzoli) dugih `DNA_SIZE`.
- Evaluiraće se kroz `GENERATIONS` generacija ili dok se ne dođe do ciljnog niza.
- Kvalitet hromozoma (fitness function, funkcija dobrote, funkcija prilagođenja) određen je udaljenošću svakog "karaktera" od odgovarajućeg ciljnog karaktera na datoj poziciji u hromozomu.
- Ide se kroz ciklični ciklus:
  - evaluacije svakog hromozoma
  - *mogao bi se se dodati i elitizam (% najboljih ide u sledeću generaciju)*
  - selekcije para hromozoma
  - ukrštanja selektovanog para hromozoma pri čemu se dobijaju dva potomka
  - mutacija potomaka
  - popunjavanje nove generacije

Simpatičan primer *Colin Drake*-a

```
# malo modifikovan primer Colin Drake
import random
OPTIMAL     = "Hello, world"
DNA_SIZE    = len(OPTIMAL)
POP_SIZE    = 40
GENERATIONS = 600

def selection(items): # npr. slucajna selekcija prema kumul. dobrot.
  n = random.uniform(items[0][1], items[len(items)-1][1])
  for strI, fitI in items:
    if fitI<=n:
      return (strI,fitI)

def random_char():  return chr(random.randint(32, 126))

def random_population():# kreiraj listu parova (string,dobrota)
  pop = []
  for i in range(POP_SIZE):
    dna = ""
    for c in range(DNA_SIZE):
      dna += random_char()
    pop.append( (dna,0) ) # neka je privremeno dobrota 0
  return pop
```

Jednostavni GA

```
def fitness(dna): # funcija dobrote (prilagodljivosti) udaljenost od cilja
  fitness = 0
  for c in range(DNA_SIZE):
    fitness += abs(ord(dna[c]) - ord(OPTIMAL[c]))
  return fitness

def mutate(dna): # izbegavanje zarobljavanja u lokalnom "dobrom" resenju
  mutstr = ""
  mutation_chance = 0.01
  for c in range(DNA_SIZE):
    if (random.random() <= mutation_chance):
        mutstr += random_char()
    else:
        mutstr += (dna[0])[c]
  return (mutstr, dna[1])

def crossover(dna1, dna2): # ukrstanje u jednoj tacki npr. AB,CD→AD,CB
  pos = int(random.random()*DNA_SIZE)
  return ( (dna1[0])[:pos] + (dna2[0])[pos:], dna1[1]) , \
         ( (dna2[0])[:pos] + (dna1[0])[pos:], dna2[1])

def pofitnessu(par): return par[1] # za list.sort(key=pofitnessu,reverse=False)
```

Jednostavni GA

```
# generisanje inicijalne populacije: lista parova (slucajnistring, dobrota)
population = random_population()
new_population = [] #sledeca generacija
# simulacija svih generacija
for generation in range(GENERATIONS):
    # pridruzivanje dobrote hromozomu
    for i in range(len(population)):
        stringI, fitnessI = population[i]
        fitnessI = fitness(stringI)
        if fitnessI == 0: # idealan hromozom
            print("GEN:%05d  Hromozom: %s Fitness: %05d" %
(generation,stringI,fitnessI))
            exit(0);
        else:
            population[i] = (population[i][0], fitnessI)

    # sortiranje populacije po dobroti
    population.sort(key=pofitnessu, reverse=False);

    #ispisi najbolji hromozom
    print('GEN:{:05d}  Hromozom: {} Fitness:
{:05d}'.format(generation,population[0][0], population[0][1]))
```

Jednostavni GA

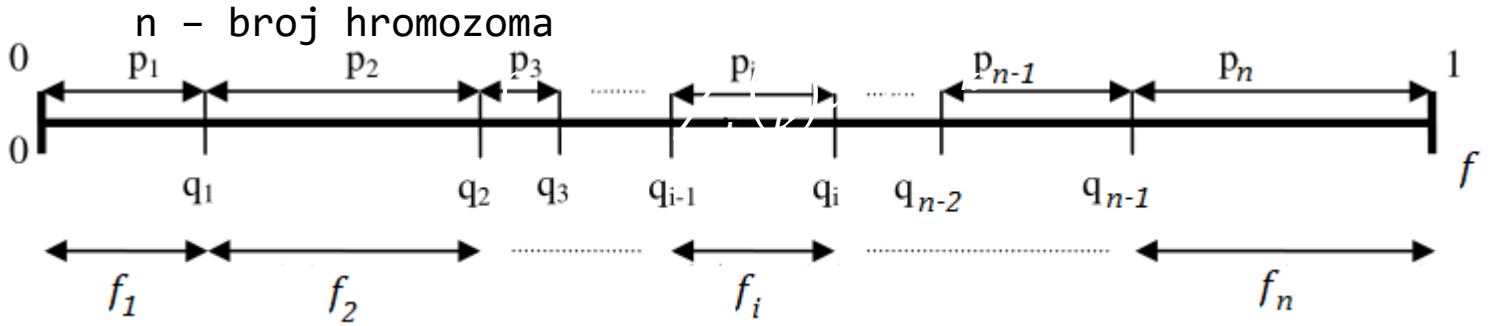
```
# postavljanje kumulativne dobrote
cumul = 0
for i in range(len(population)):
    cumul += population[i][1]
    population[i] = (population[i][0], cumul)

for _ in range(POP_SIZE//2):
    # selekcija roditelja
    par1 = selection(population)
    par2 = selection(population)

    # ukrstanje roditelja i stvaranje dva potpomka
    child1, child2 = crossover(par1, par2)

    # mutacija i punjenje sledece generacije
    new_population.append(mutate(child1))
    new_population.append(mutate(child2))

 new_population.sort(key=pofitnessu, reverse=False);

 population = new_population
 new_population = []
```

Izlaz

Ukupna dobrota $f = \sum_{i=1}^{n} f_i$

Kumulativna dobrota hromozoma k, $q_k = \sum_{i=1}^{k} f_i$

Verovanoća selekcije hromozoma k, $p_k = \frac{f_k}{f}$



Ovo koristiti u slučajevima kada je dobrota bolja ako je njena vrednost veća.

```
GEN:00001  Hromozom: _ft\n98fop\b Fitness: 00121
GEN:00002  Hromozom: _ft\n98fopdb Fitness: 00113
...
GEN:00534  Hromozom: Hello,!world Fitness: 00001
GEN:00535  Hromozom: Hello, world Fitness: 00000
```

Za domaći : poboljšati dati algoritam.