Programarea calculatoarelor si limbaje de programare II

Administrarea atributelor
Decoratori
Metaclase

Universitatea Politehnica din Bucureşti

Sumar

- ***Cod asociat cu accesarea atributelor***
- Proprietati
- Descriptori
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- Definitia decoratorilor
- Decorarea functiilor
- Decorarea claselor

Tehnici de asociere

1. Metodele __***getattr***__ (pentru citirea atributelor nedefinite) si __***setattr***__ (scriere atribute), in orice versiune de Python
2. Metoda __***getattribute***__ (toate citirile de atribute)
3. Proprietati – protocol predefinit
4. Descriptori – protocol bazat pe metodele __***get***__ si __***set***__ din instante de clase arbitrare

Sumar

- Cod asociat cu accesarea atributelor
- ***Proprietati***
- Descriptori
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- Definitia decoratorilor
- Decorarea functiilor
- Decorarea claselor

Sintaxa proprietatilor

- O proprietate se creeaza prin atribuirea:

```
attribute = property(fget, fset, fdel, doc)
```

  - unde *attribute* este un atribut de clasa, iar *property* este (o clasa) predefinita din modulul *builtins*
  - *fget* este functia folosita la interceptarea operatiilor de citire a lui *attribute*
  - *fset* – scrierea lui *attribute*
  - *fdel* – stergerea lui *attribute*
  - *doc* – docstring pentru *attribute* (implicit docstringul lui *fget*)

- Exemplu:

```
class Person: # Cu (object) in v2.X
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('fetch...')
        return self._name
    def setName(self, value):
        print('change...')
```

Sintaxa...

```
            self._name = value
        def delName(self):
            print('remove...')
            del self._name
        name = property(getName, setName,
        delName, "name property docs") # Atribut
        de clasa, name, se mosteneste!
bob = Person('Bob Smith') # Instanta bob are
    atributul name, administrat
print(bob.name) # Apel de getName
bob.name = 'Robert Smith' # Apel de setName
print(bob.name)
del bob.name # Apel de delName
print('-'*20)
sue = Person('Sue Jones') # Instanta sue
    mosteneste de asemenea proprietatea name
print(sue.name)
print(Person.name.__doc__) # Sau
    help(Person.name)
```

```
C:\code>py -3 prop-person.py
fetch...
Bob Smith
change...
fetch...
Robert Smith
remove...
--------------------
fetch...
Sue Jones
name property docs
```

Codificare cu decoratori

- Sintaxa decoratorilor de tip functie:

```
@decorator                          def func(args): ... # Cod echivalent
def func(args): ...                 func = decorator(func)
```

- **property** se poate folosi ca decorator
  - La fel: **getter** (creat automat cu *property*), **setter**, **deleter**

```
class Person:
    def __init__(self, name):
        self._name = name
    @property
    def name(self): # name = property(name)
        "name property docs"
        print('fetch...')
        return self._name
    @name.setter
    def name(self, value): # name = name.setter(name)
        print('change...')
        self._name = value
    @name.deleter
    def name(self): # name = name.deleter(name)
        print('remove...')
        del self._name
```

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- ***Descriptori***
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- Definitia decoratorilor
- Decorarea functiilor
- Decorarea claselor

Sintaxa descriptorilor

- Descriptorii sunt si ei folositi la interceptarea accesului la atribute
  - De fapt, proprietatile sunt un tip particular de descriptori
- Descriptorii sunt implementati ca **clase** – ce au metodele **__get__**, **__set__**, si **__delete__**

```
class Descriptor:
    "docstring goes here"
    def __get__(self, instance, owner): ... # Returneaza valoarea atributului
    def __set__(self, instance, value): ... # Returneaza None
    def __delete__(self, instance): ... # Returneaza None
```

  - Argumentul *self* este instanta descriptorului
  - Argumentul *instance* este instanta clasei (client) care foloseste descriptorul
  - Argumentul *owner* este clasa client

Sintaxa...

- Exemplu:

```
>>> class Descriptor: # Cu (object) in v2.X
        def __get__(self, instance, owner):
            print(self, instance, owner, sep='\n')

>>> class Client:# Cu (object) in v2.X
        attr = Descriptor() # Atribut de clasa, attr, egal cu o instanta a clasei Descriptor

>>> X = Client()
>>> X.attr
```

```
<__main__.Descriptor object at 0x000001F7A625DE88>
<__main__.Client object at 0x000001F7A5F92708>
<class '__main__.Client'>
>>> Client.attr # Argumentul instance este None!
<__main__.Descriptor object at 0x000001F7A625DE88>
None
<class '__main__.Client'>
```

  - Cod echivalent: **X.attr -> Descriptor.__get__(Client.attr, X, Client)**

Descriptori *read-only*

- Pentru a se impiedica scrierea unui descriptor, metoda *__set__* din clasa descriptor trebuie sa declanseze o exceptie:

```
>>> class D:
        def __get__(*args): print('get')
        def __set__(*args): raise
      AttributeError('cannot set')


>>> class C:
        a = D()
```

```
>>> X = C()
>>> X.a # Apel de C.a.__get__
get
>>> X.a = 99 # Apel de C.a.__set__
AttributeError: cannot set
```

Exemplu

- Atributul *name* din clasa *Person*:

```
class Name: # Cu (object) in v2.x
    "name descriptor docs"
    def __get__(self, instance, owner):
        print('fetch...')
        return instance._name
    def __set__(self, instance, value):
        print('change...')
        instance._name = value
    def __delete__(self, instance):
        print('remove...')
        del instance._name
class Person: # Cu (object) in 2.X
    def __init__(self, name):
        self.name = name
    name = Name() # Atribuire a descriptorului Name() catre atributul name
bob = Person('Bob Smith') # bob.name este atribut administrat
print(bob.name) # Se executa Name.__get__
bob.name = 'Robert Smith' # Se executa Name.__set__
print(bob.name)
del bob.name # Se executa Name.__delete__
print('-'*20)
sue = Person('Sue Jones') # sue are atribut mostenit
print(sue.name)
print(Name.__doc__) # Sau help(Name)
```

Exemplu...

- Cu descriptor (clasa *Name*) inclus in clasa *Person*:

```
class Person:
    def __init__(self, name):
        self.name = name
    class Name: # Clasa inclusa/imbricata
        "name descriptor docs"
        def __get__(self, instance, owner):
            print('fetch...')
            return instance._name
        def __set__(self, instance, value):
            print('change...')
            instance._name = value
        def __delete__(self, instance):
            print('remove...')
            del instance._name
    name = Name()
```

Atribut calculat

- Descriptor folosit la evaluarea atributului:

```
class DescSquare: # Fara delete si fara docstring
    def __init__(self, start): # start reprezinta informatie de stare
        self.value = start
    def __get__(self, instance, owner):
        return self.value ** 2
    def __set__(self, instance, value):
        self.value = value

class Client1:
    X = DescSquare(3)

class Client2:
    X = DescSquare(32)

c1 = Client1()
c2 = Client2()

print(c1.X) # 3 ** 2
c1.X = 4
print(c1.X) # 4 ** 2
print(c2.X) # 32 ** 2 (1024)
```

```
c:\code> py –3 desc-computed.py
9
16
1024
```

Informatii de stare in descriptori

- Starea poate fi memorata cu o combinatie de:
  - **Stare de descriptor** – asociata cu clasa descriptorului
  - **Stare de instanta** – este creata de o instanta a clasei client (care foloseste descriptorul)
- Stare de descriptor:

```
class DescState: # Descriptor – cu stare
    def __init__(self, value):
        self.value = value
    def __get__(self, instance, owner):
        print('DescState get')
        return self.value * 10
    def __set__(self, instance, value):
        print('DescState set')
        self.value = value
```

```
class CalcAttrs: # Clasa client
    X = DescState(2) # Atribut cu descriptor
    Y = 3 # Atribut de clasa
    def __init__(self):
        self.Z = 4 # Atribut de instanta

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z) # X este atribut calculat
```

Informatii...

```
obj.X = 5 # Atribuirea este interceptata
CalcAttrs.Y = 6 # Y este reasignat la nivel de clasa
obj.Z = 7 # Z reasignat la nivel de instanta
print(obj.X, obj.Y, obj.Z)
obj2 = CalcAttrs() # X este partajat, la fel ca Y !
print(obj2.X, obj2.Y, obj2.Z)
```

```
c:\code> py –3 desc-state-desc.py
DescState get
20 3 4
DescState set
DescState get
50 6 7
DescState get
50 6 4
```

- Stare de instanta a clientului:

```
class InstState: # Instanta – cu stare
    def __get__(self, instance, owner):
        print('InstState get') # instance este a clasei client!
        return instance._X * 10
    def __set__(self, instance, value):
        print('InstState set')
        instance._X = value

class CalcAttrs: # Clasa client
    X = InstState() # Atribut cu descriptor
    Y = 3 # Atribut de clasa
```

Informatii...

```
    def __init__(self):
        self.X = 2 # Setare via descriptor
        self.Z = 4 # Atribut de instanta

obj = CalcAttrs()
print(obj.X, obj.Y, obj.Z) # X este atribut calculat
obj.X = 5 # Atribuire interceptata
CalcAttrs.Y = 6 # Reasignare in clasa
obj.Z = 7 # Reasignare in instanta
print(obj.X, obj.Y, obj.Z)

obj2 = CalcAttrs() # X este diferit, acum, ca Z!
print(obj2.X, obj2.Y, obj2.Z)
```

```
c:\code> py –3 desc-state-inst.py
InstState set
InstState get
20 3 4
InstState set
InstState get
50 6 7
InstState set
InstState get
20 6 4
```

Informatii...

- Stare atat de descriptor cat si de instanta:

```
>>> class DescBoth:
        def __init__(self, data):
            self.data = data
        def __get__(self, instance, owner):
            return '%s, %s' % (self.data,
    instance.data)
        def __set__(self, instance, value):
            instance.data = value
>>> class Client:
        def __init__(self, data):
            self.data = data
        managed = DescBoth('spam')
```

```
>>> I = Client('eggs')
>>> I.managed # Atribut cu stare multipla
'spam, eggs'
>>> I.managed = 'SPAM' # Schimbare la nivel de
    instanta
>>> I.managed
'spam, SPAM'
```

property definita cu descriptori

- Clasa *Property* simuleaza clasa predefinita *property*:

```
class Property:
    def __init__(self, fget=None, fset=None,
fdel=None, doc=None): # Se salveaza
metodele si alte atribute
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
    def __get__(self, instance,
instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("can't get
attribute")
        return self.fget(instance)
def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set
attribute")
        self.fset(instance, value)
def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't
delete attribute")
        self.fdel(instance)
```

property…

```
class Person:
    def getName(self):print('getName...')
    def setName(self, value):print('setName...')
    # Property este folosita ca property!
    name = Property(getName, setName)
```

```
x = Person()
x.name
x.name = 'Bob'
del x.name
```

```
c:\code> py –3 prop-desc-equiv.py
getName...
setName...
AttributeError: can't delete attribute
```

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- Descriptori
- ***__getattr__ si __getattribute__***
- Validarea atributelor
- Definitia decoratorilor
- Decorarea functiilor
- Decorarea claselor

Sintaxa

- ***__getattr__*** este folosit pentru atribute nedefinite
- ***__getattribute__*** este folosit pentru orice atribut
- Folosesc protocolul de inlocuire a operatorilor
- Pentru interceptarea asignarilor se foloseste ***__setattr__*** iar stergerea atributelor se face cu ***__delattr__***

```
def __getattr__(self, name): # Pentru citirea atributului nedefinit [obj.name]
def __getattribute__(self, name): # Pentru citirea oricarui atribut [obj.name]
def __setattr__(self, name, value): # Pentru asignarile atributului [obj.name=value]
def __delattr__(self, name): # Pentru stergerea atributului [del obj.name]
```

```
class Catcher: # __getattr__ si __setattr__
    def __getattr__(self, name):
        print('Get: %s' % name)
    def __setattr__(self, name, value):
        print('Set: %s %s' % (name, value))
```

```
X = Catcher()
X.job # Afiseaza "Get: job"
X.pay # Afiseaza "Get: pay"
X.pay = 99 # Afiseaza "Set: pay 99"
```

Sintaxa...

```
class Catcher(object): # Cu (object) doar in v2.x
    def __getattribute__(self, name):
        print('Get: %s' % name) # Atentie la recursivitate!
...restul este la fel...
```

```
class Wrapper: # Delegatie!
    def __init__(self, object):
        self.wrapped = object
    def __getattr__(self, attrname):
        print('Trace: ' + attrname)
        return getattr(self.wrapped, attrname) # Delegatie catre wrapped
```

```
X = Wrapper([1, 2, 3])
X.append(4) # Afiseaza "Trace: append"
print(X.wrapped) # Afiseaza "[1, 2, 3, 4]"
```

Pericolul recursivitatii

- Apare la folosirea lui *__getattribute__* si *__setattr__*:

```
def __getattribute__(self, name):
    x = self.other # Recursivitate infinita!
```

- Evitarea recursivitatii infinite, prin folosirea unei superclase - *object*:

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other') # Se forteaza apel via superclasa
```

- La fel pentru *__setattr__*:

```
def __setattr__(self, name, value):
    self.other = value # Recursivitate (infinita)
```

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value # Recursivitate evitata cu __dict__
```

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value) # Evitare cu superclasa object
```

Pericolul...

- In cazul lui __***delattr***__ recursivitatea infinita se evita tot cu folosirea lui __*dict*__ sau a unei superclase (*object*)

Exemplu

- Atributul *name* din clasa *Person*, cu comparatii:

```
class Person: # Portabil, in v2.x sau v3.x
    def __init__(self, name): # Constructor
        self.name = name # Apel de __setattr__!
    def __getattr__(self, attr): # Pentru [obj.undefined]
        print('get: ' + attr)
        if attr == 'name': # Cazul 'name'
            return self._name # Fara recursivitate, _name exista!
        # Alte atribute sunt erori
        raise AttributeError(attr)
    def __setattr__(self, attr, value): # Pentru [obj.any = value]
        print('set: ' + attr)
        if attr == 'name':
            attr = '_name' # Numele intern
        self.__dict__[attr] = value # Evitarea recursivitatii cu __dict__
    def __delattr__(self, attr): # Pentru [del obj.any]
        print('del: ' + attr)
        if attr == 'name':
            attr = '_name'
        del self.__dict__[attr] # Recursivitate evitata
```

Exemplu...

```
bob = Person('Bob Smith')
print(bob.name) # Apel de __getattr__
bob.name = 'Robert Smith' # Apel de
    __setattr__
print(bob.name)
del bob.name # Apel de  __delattr__
print('-'*20)
sue = Person('Sue Jones')
print(sue.name)
```

```
c:\code> py –3 getattr-person.py
set: name
get: name
Bob Smith
set: name
get: name
Robert Smith
del: name
--------------------
set: name
get: name
Sue Jones
```

Exemplu...

- Exemplu cu atribute evaluate (X):

```
class AttrSquare:
    def __init__(self, start):
        self.value = start # Apel de __setattr__!
    def __getattr__(self, attr): # Pentru atribute nedefinite
        if attr == 'X':
            return self.value ** 2 # Evaluare
        raise AttributeError(attr)
def __setattr__(self, attr, value): # Pentru orice asignare de atribut
    if attr == 'X':
        attr = 'value'
    self.__dict__[attr] = value

A = AttrSquare(3) # Instanta
B = AttrSquare(32) # Alta instanta cu alta stare
print(A.X) # 3 ** 2
A.X = 4
print(A.X) # 4 ** 2
print(B.X) # 32 ** 2 (1024)
```

```
c:\code> py –3 getattr-computed.py
9
16
1024
```

Tehnici de administrare a atributelor

- 1. Cu ***property***:

```
# Doua atribute (square si cube) evaluate
    dinamic, cu property
class Powers(object): # Cu (object) numai in v2.x
    def __init__(self, square, cube):
        self.square = square
        self.cube = cube
    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)
    def getCube(self):
        return self._cube ** 3
    def setCube(self, value):
        self._cube = value
    cube = property(getCube, setCube)

X = Powers(3, 4)
print(X.square) # 3 ** 2 = 9
print(X.cube) # 4 ** 3 = 64
X.square = 5
print(X.square) # 5 ** 2 = 25
```

Tehnici...

- 2. Cu **descriptori**:

```
# Cu descriptori, stare per instanta
class DescSquare(object):
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value
class DescCube(object):
    def __get__(self, instance, owner):
        return instance._cube ** 3
    def __set__(self, instance, value):
        instance._cube = value
```

```
class Powers(object): # Cu (object) numai in v2.x
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self.square = square
        self.cube = cube

X = Powers(3, 4)
print(X.square) # 3 ** 2 = 9
print(X.cube) # 4 ** 3 = 64
X.square = 5
print(X.square) # 5 ** 2 = 25
```

Tehnici...

- 3. Cu ***__getattr__***:

```
# Cu __getattr__ pentru atribute nedefinite
class Powers:
    def __init__(self, square, cube):
        self.square = square
        self.cube = cube
    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        if name == 'cube':
            return self._cube ** 3
        raise TypeError('unknown attr:' + name)
    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value
        # Sau cu object...
        elif name == 'cube':
            self.__dict__['_cube'] = value
        else:
            self.__dict__[name] = value

X = Powers(3, 4)
print(X.square) # 3 ** 2 = 9
print(X.cube) # 4 ** 3 = 64
X.square = 5
print(X.square) # 5 ** 2 = 25
```

Tehnici...

- 4. Cu ***__getattribute__***:

```
# Cu __getattribute__, orice atribute
class Powers(object): # Cu (object) doar in v2.x
    def __init__(self, square, cube):
        self.square = square
        self.cube = cube
    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        if name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        return object.__getattribute__(self, name)
    def __setattr__(self, name, value):
        if name == 'square':
            object.__setattr__(self, '_square', value) # Sau cu __dict__
        elif name == 'cube':
            object.__setattr__(self, '_cube', value)
        else:
            object.__setattr__(self, name , value)
X = Powers(3, 4)
print(X.square) # 3 ** 2 = 9
print(X.cube) # 4 ** 3 = 64
X.square = 5
print(X.square) # 5 ** 2 = 25
```

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- Descriptori
- *__getattr__* si *__getattribute__*
- ***Validarea atributelor***
- Definitia decoratorilor
- Decorarea functiilor
- Decorarea claselor

Validare cu ***property***

- Se folosesc atribute redenumite (*__name*):

```
# Fisier validate_properties.py:
class CardHolder(object): # Cu (object) in v2.x pentru setter
    acctlen = 8 # Atr. de clasa
    retireage = 59.5 # Atr. de clasa
    def __init__(self, acct, name, age, addr):
        self.acct = acct # Atr. de instanta
        self.name = name # Apel de setter!
        self.age = age # __X redenumit cu numele clasei
        self.addr = addr # Atr. neadministrat
    def getName(self):
        return self.__name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self.__name = value
    name = property(getName, setName)
    def getAge(self):
        return self.__age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('invalid age')
        else:
            self.__age = value
    age = property(getAge, setAge)
```

Validare...

```
def getAcct(self):
        return self.__acct[:-3] + '***'
def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
                raise TypeError('invald acct
number')
        self.__acct = value
acct = property(getAcct, setAcct)
```

```
def remainGet(self): # Poate fi chiar si o
metoda daca nu este folosita deja ca atribut
        return self.retireage - self.age
remain = property(remainGet)
```

Validare cu descriptori

- Cu stare partajata in instanta descriptorului:

```
# Fisier validate_descriptors1.py:
class CardHolder(object): # Cu (object) doar in
    v2.x
    acctlen = 8
    retireage = 59.5
    def __init__(self, acct, name, age, addr):
        self.acct = acct # Atribut de instanta
        self.name = name # Apel de __set__!
        self.age = age # Idem
        self.addr = addr # Neadministrat
    class Name(object):  # Descriptor inclus
        def __get__(self, instance, owner):
            return self.name
        def __set__(self, instance, value):
            value = value.lower().replace('
', '_')
            self.name = value
name = Name()
class Age(object): # Descriptor inclus
    def __get__(self, instance, owner):
        return self.age
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('invalid
age')
        self.age = value
age = Age()
```

Validare...

```
class Remain(object): # Descriptor inclus
        def __get__(self, instance, owner):
            return instance.retireage - instance.age # Cu apeluri de Age.__get__
        def __set__(self, instance, value):
            raise TypeError('cannot set remain')
remain = Remain()
```

- Cu stare per client (ce foloseste descriptorul) si __X (mangling):

```
# Fisier validate_descriptors2.py:
class CardHolder(object): # Cu (object) doar in v2.x
    acctlen = 8 # Atribute de clasa
    retireage = 59.5
    def __init__(self, acct, name, age, addr):
        self.acct = acct # Atribute de client
        self.name = name # Apel de __set__!
        self.age = age # Idem
        self.addr = addr # Neadministrat
```

Validare...

```
class Name(object): # Descriptor inclus
        def __get__(self, instance, owner):
                return instance.__name
        def __set__(self, instance, value):
                value = value.lower().replace(' ', '_')
                instance.__name = value
name = Name() # Atribut administrat

class Age(object): # Descriptor, inclus
        def __get__(self, instance, owner):
                return instance.__age
        def __set__(self, instance, value):
                if value < 0 or value > 150:
                        raise ValueError('invalid age')
                instance.__age = value
age = Age() # Atribut administrat

class Acct(object): # Descriptor inclus
        def __get__(self, instance, owner):
                return instance.__acct[:-3] + '***'
        def __set__(self, instance, value):
                value = value.replace('-', '')
                if len(value) != instance.acctlen:
                        raise TypeError('invald acct number')
                instance.__acct = value
```

Validare...

```
acct = Acct() # Atribut administrat


class Remain(object): # Descriptor inclus
        def __get__(self, instance, owner):
                return instance.retireage - instance.age # Apel de Age.__get__
        def __set__(self, instance, value):
        raise TypeError('cannot set remain') # Fara date
remain = Remain() # Atribut administrat
```

Validare cu ***__getattr__***

- ***__getattr__*** intercepteaza atributele nedefinite:

```
# Fisier validate_getattr.py:
class CardHolder:
    acctlen = 8 # Atribut de clasa
    retireage = 59.5
    def __init__(self, acct, name, age, addr):
        self.acct = acct # Inlocuit cu _acct
        self.name = name # Apel de __setattr__!
        self.age = age # Idem
        self.addr = addr # Neadministrat
    def __getattr__(self, name):
        if name == 'acct':
            return self._acct[:-3] + '***'
        if name == 'remain':
            return self.retireage - self.age # Fara recursivitate de __getattr__
        raise AttributeError(name)
    def __setattr__(self, name, value):
        if name == 'name':
            value = value.lower().replace(' ', '_')
        elif name == 'age':
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        elif name == 'acct':
```

Validare...

```
            name = '_acct'
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invald acct number')
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value # Se evita recursivitatea (sau cu object)
```

Validare cu ***__getattribute__***

- ***__getattribute__*** intercepteaza toate citirile de atribute:

```
# Fisier validate_getattribute.py:
class CardHolder(object): # Cu (object) doar in v2.x
    acctlen = 8 # Atribute de clasa
    retireage = 59.5
    def __init__(self, acct, name, age, addr):
        self.acct = acct # Atribute de instanta
        self.name = name # Apeluri de __setattr__!
        self.age = age
        self.addr = addr # Atr. neadministrat

    def __getattribute__(self, name):
        superget = object.__getattribute__ # Se evita recursivitatea infinita
        if name == 'acct':
            return superget(self, 'acct')[:-3] + '***'
        if name == 'remain':
            return superget(self, 'retireage') - superget(self, 'age')
        return superget(self, name)

def __setattr__(self, name, value):
    if name == 'name':
        value = value.lower().replace(' ', '_')
    elif name == 'age':
```

Validare...

```
            if value < 0 or value > 150:
                raise ValueError('invalid age')
        elif name == 'acct':
            value = value.replace('-', '')
            if len(value) != self.acctlen:
                raise TypeError('invald acct number')
        elif name == 'remain':
            raise TypeError('cannot set remain')
        self.__dict__[name] = value # Evitarea recursivitatii infinite cu __dict__
```

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- Descriptori
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- ***Definitia decoratorilor***
- Decorarea functiilor
- Decorarea claselor

Sintaxa decoratorilor

- Definitie: decorarea (@decorator) inseamna o modalitate de administrare sau de extindere a functiilor si claselor in Python.
- Categorii de decoratori:
  - Decoratori de functii si metode
  - Decoratori de clase
- Pentru ambele categorii, decoratorul insereaza cod care este executat automat la sfarsitul definitiei functiei sau clasei.
- Utilizari ale decoratorilor – pentru extinderea apelurilor :
  - Proxy – intermediar – pentru apeluri ulterioare de functii
  - Proxy pentru interfete, la crearea ulterioara a instantei clasei

Sintaxa...

- Utilizari ale decoratorilor – pentru administrarea functiilor si claselor:
  - Administrarea obiectelor de tip functie – e.g delegatie
  - Administrarea obiectelor de tip clasa – e.g. adaugarea de metode noi
- Programarea cu decoratori:
  - Ca utilizator sau creator de decoratori – predefiniti, e.g. metode statice si de clasa, proprietati.
  - Cod pentru trasarea sau logarea executiei functiilor
  - Cod pentru verificarea validitatii argumentelor – in faza de depanare
  - Decoratorii de clase pot extinde interfata unei clase, pot trasa, valida referintele la atributele clasei, pot crea clase de tip *singleton*, cu tehnici de delegatie

Sintaxa...

- Avantajele decoratorilor:
  - Au o sintaxa explicita
  - Se aplica o singura data la definitia functiei sau a clasei decorate
  - Asigura administrarea simpla si consistenta a codului
- Dezavantaj: pot altera tipul obiectelor decorate

Decoratori de functii

- Se utilizeaza incepand cu versiunea v2.4 de Python:

```
@decorator # Decorarea functiei
def F(arg):
...
F(99) # Apelul functiei
```

```
# Cod echivalent:
def F(arg):
...
F = decorator(F) # Redefinirea numelui functiei
    ca fiind rezultatul apelului decoratorului
F(99) # Practic, este apel de decorator(F)(99)
```

- Exemple cunoscute – metode statice si decorarea cu *property*:

```
class C:
    @staticmethod
    def meth(...): ... # meth =
    staticmethod(meth)
```

```
class C:
    @property
    def name(self): ... # name = property(name)
```

Decoratori..

- Implementare - obiect apelabil care returneaza un obiect apelabil:

```
def decorator(F):
    # Procesarea functiei F
    return F
```

```
@decorator
def func(): ... # func = decorator(func)
```

- Obiectul returnat poate diferi de functia decorata:

```
def decorator(F):
    # Functia F este memorata sau/si folosita
    # Se returneaza alt obiect apelabil: un def inclus, sau un class cu __call__, etc.
```

```
@decorator
def func(): ... # func = decorator(func)
```

- Decorarea functiei cu o functie:

```
def decorator(F): # Pentru decorare cu @
    def wrapper(*args): # Pentru apelul functiei decorate
        # Folosirea lui F si args
        # F(*args) apeleaza functia originala
    return wrapper
@decorator # func = decorator(func)
def func(x, y): ... # func e F in decorator
func(6, 7) # 6, 7 sunt transmise cu *args
```

Decoratori..

- Decorarea functiei cu un decorator de tip clasa:

```
class decorator:
    def __init__(self, func): # Constructor apelat la momentul decorarii cu @decorator
        self.func = func # Atribut care retine functia decorata
    def __call__(self, *args): # Metoda apelata la apelul functiei decorate
        # Se foloseste self.func si args
        # self.func(*args) este apel al functiei decorate
```

```
@decorator
def func(x, y): # func = decorator(func)
    ... # func e argument pentru __init__
func(6, 7) # 6, 7 sunt *args ale lui __call__
```

- Decorarea **metodelor** cu o clasa – **NU merge**
  - Deoarece instanta metodei decorate NU este pasata in *args, ceea ce face imposibil apelul metodei originale:

Decoratori..

```
class decorator:
    def __init__(self, func): # func este metoda fara instanta!
        self.func = func
    def __call__(self, *args): # self este instanta decoratorului
        # self.func(*args) nu merge deoarece instanta  lui C lipseste din *args
class C:
    @decorator
    def method(self, x, y): ... # method = decorator(method)
```

- Corect – decorarea metodei/functiei cu o functie:

```
def decorator(F): # F e functie sau metoda
    def wrapper(*args): # Instanta clasei este args[0] in cazul metodei
        # F(*args) apeleaza functia sau metoda
        return wrapper
@decorator
def func(x, y): ... # func = decorator(func)
func(6, 7) # Apel de wrapper(6, 7)
class C:
    @decorator
    def method(self, x, y): ... # method = decorator(method)
X = C()
X.method(6, 7) # Apel de wrapper(X, 6, 7)
```

Decoratori de clase

- Se utilizeaza incepand cu versiunea v2.6 si v3.0:

```
@decorator # Decorarea clasei
class C:
    ...
x = C(99) # Instanta a clasei C
```

```
# Cod echivalent:
class C:
    ...
C = decorator(C) # Numele clasei refera
    rezultatul apelului de decorator
x = C(99) # Practic,  apel de decorator(C)(99)
```

- Administrarea clasei, cu returnarea clasei decorate:

```
def decorator(C):
    # Procesarea clasei C
    return C
```

```
@decorator
class C: ... # C = decorator(C)
```

- Decorarea clasei, cu returnarea unui alt obiect apelabil:

```
def decorator(C):
    # Memoreaza sau/si apeleaza clasa C
    # Returneaza: alt obiect inclus, un def, sau
      class cu __call__, etc.
```

```
@decorator
class C: ... # C = decorator(C)
```

Decoratori...

- Clasa inclusa, care intercepteaza atributele nedefinite:

```
def decorator(cls): # Decorator – functie
    class Wrapper: # Clasa inclusa
        def __init__(self, *args): # Constructor de instanta
            self.wrapped = cls(*args) # Instanta a clasei decorate
        def __getattr__(self, name): # Pentru citirea atributelor
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C: # C = decorator(C)
    def __init__(self, x, y): # Apelat de Wrapper.__init__
        self.attr = 'spam'

x = C(6, 7) # Practic apel de Wrapper(6, 7)
print(x.attr) # Apel de Wrapper.__getattr__, afiseaza "spam"
```

- Decoratorul de clasa poate fi o fabrica de functii sau de clase avand metodele *__init__* sau *__call__*

Decoratori...

- Decorarea clasei cu un decorator de tip clasa **NU** merge:

```
class Decorator:
    def __init__(self, C): # Apelat la decorare
        self.C = C
    def __call__(self, *args): # Apelat la crearea instantei clasei decorate
        self.wrapped = self.C(*args) # Acelasi atribut pentru instante diferite, ultima conteaza
        return self
    def __getattr__(self, attrname): # Pentru citirea atributelor
        return getattr(self.wrapped, attrname)

@Decorator
class C: ... # C = Decorator(C), o singura instanta a clasei Decorator!
x = C()
y = C() # Suprascrierea atributului wrapped...
```

- Corect: cu includerea instantei clasei decorate intr-o instanta noua a decoratorului:

Decoratori...

 # Cu clasa inclusa

```
def decorator(C): # Decorator, cu @
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = C(*args) # Instanta noua
    return Wrapper
```

 # Cu functie inclusa, clasa externa

```
class Wrapper: ...
def decorator(C): # Decorator, cu @
    def onCall(*args): # La crearea instantei, un nou Wrapper
        return Wrapper(C(*args)) # Instanta noua
    return onCall
```

Decoratori multipli/inclusi

- Decorarea functiilor cu mai multi decoratori, imbricati:

```
@A                          # Cod echivalent:
@B                          def f(...):
@C                              ...
def f(...):                 f = A(B(C(f)))
    ...
```

- Decorarea claselor cu mai multi decoratori, imbricati:

```
@spam                       # Cod echivalent:
@eggs                       class C:
class C:                        ...
    ...                     C = spam(eggs(C))
X = C()                     X = C()
```

Decoratori...

- Exemplu, decoratori multipli de functie:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func(): # func = d1(d2(d3(func)))
    return 'spam'


print(func()) # Afiseaza "XYZspam"
```

Decoratori cu argumente

- Sintaxa:

```
@decorator(A, B)
def F(arg):
    ...
F(99)
```

```
# Cod echivalent:
def F(arg):
    ...
F = decorator(A, B)(F) # F este rezultatul
    returnat de decorator
F(99) # Apel de: decorator(A, B)(F)(99)
```

- Argumentele decoratorului pot fi memorate si folosite ulterior:

```
def decorator(A, B):
    # Salvare/utilizare A, B
    def actualDecorator(F):
        # Salvare/utilizare F
        # Returnarea unui obiect apelabil
        inclus: un def, un class cu __call__, etc.
        return callable
    return actualDecorator
```

- Rolul argumentelor: valori de initializare a atributelor, trasarea executiei, etc.

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- Descriptori
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- Definitia decoratorilor
- ***Decorarea functiilor***
- Decorarea claselor

Trasarea apelurilor

- Contorizare si trasare a apelurilor unei functii:

*# Fisier **decorator1.py**:*

```
class tracer:
    def __init__(self, func): # La decorare se salveaza functia decorata
        self.calls = 0 # Initializare contor
        self.func = func
    def __call__(self, *args): # La apeluri ulterioare
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__)) # Trasare
        self.func(*args)  # Apelul functiei decorate

@tracer # spam este decorata
def spam(a, b, c): # spam = tracer(spam)
    print(a + b + c)
```

```
C:\code>py -3
>>> from decorator1 import spam
>>> spam(1, 2, 3) #Apel trasat
call 1 to spam
6
>>> spam('a', 'b', 'c') # Apel de __call__
abc
>>> spam.calls # Valoare contor
2
>>> spam
<decorator1.tracer object at 0x00000171204C4C48>
```

Tehnici de memorare a starii cu decoratori

- Cu atribute ale instantei decoratorului:

```
class tracer: # Stare via atribute de instanta
    def __init__(self, func): # Apelat La decorare
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs): # Apelat la apelul functiei decorate
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

@tracer
def spam(a, b, c): # spam = tracer(spam), se executa tracer.__init__
    print(a + b + c)

@tracer
def eggs(x, y): # eggs = tracer(eggs)
    print(x ** y)

spam(1, 2, 3) # Apel de tracer.__call__
spam(a=4, b=5, c=6)
eggs(2, 16) # Acum self.func este eggs
eggs(4, y=4) # Contor diferit, per decoratie
```

Tehnici...

- Cu variabile globale si declaratia *global*:

```
calls = 0 # Contor global, unic

def tracer(func):
    def wrapper(*args, **kwargs):
        global calls
        calls += 1
        print('call %s to %s' % (calls,
func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c): # spam = tracer(spam)
    print(a + b + c)
@tracer
def eggs(x, y): # eggs = tracer(eggs)
    print(x ** y)


spam(1, 2, 3)
spam(a=4, b=5, c=6)
eggs(2, 16)
eggs(4, y=4)
```

Tehnici...

- Cu domeniu inclus si declaratia *nonlocal* – in v3.x:

```
def tracer(func): # Contor diferit
    calls = 0 # Stare per functie/decorator
    def wrapper(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer # spam = tracer(spam)
def spam(a, b, c):
    print(a + b + c)
# Etc.
```

Tehnici...

- Cu domeniu inclus si atribute de functie:

```
def tracer(func): # Fiecare apel de tracer produce un wrapper diferit!
    def wrapper(*args, **kwargs):
        wrapper.calls += 1 # Contor diferit, per functie
        print('call %s to %s' % (wrapper.calls, func.__name__))
        return func(*args, **kwargs)
    wrapper.calls = 0 # Initializare atribut de functie – contor
    return wrapper

@tracer # spam = tracer(spam)
def spam(a, b, c):
    print(a + b + c)
# Etc.
```

Decorarea metodelor

- Cu functii incluse:

```
def tracer(func): # class cu __call__ este incorect! Deci, cu functii!
    calls = 0
    def onCall(*args, **kwargs): # Sau cu onCall.calls += 1
        nonlocal calls
        calls += 1
        print('call %s to %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall
```

```
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    @tracer
    def giveRaise(self, percent): # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent) # onCall a retinut pe giveRaise
    @tracer
    def lastName(self): # lastName = tracer(lastName)
        return self.name.split()[-1]
```

Decorarea...

- Cu descriptori – in v3.x:

```
class Descriptor(object):
    def __get__(self, instance, owner): ...

class Subject:
    attr = Descriptor()

X = Subject()
X.attr # Apel de Descriptor.__get__(Subject.attr, X, Subject)
```

Decorarea...

- Cu decorator de tip descriptor – in v3.x:

```
class tracer(object): # Este decorator+descriptor
    def __init__(self, func): # La decorare
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs): # La apelul functiei decorate
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner): # La citirea atributului
        return wrapper(self, instance)
class wrapper:
    def __init__(self, desc, subj): # Retine ambele instante
        self.desc = desc
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs) # Apel de tracer.__call__
@tracer
def spam(a, b, c): # spam = tracer(spam)
    ...la fel... # Foloseste doar pe __call__
class Person:
    @tracer
    def giveRaise(self, percent): # giveRaise = tracer(giveRaise)
        ...la fel... # giveRaise devine descriptor
```

Masurarea timpului de executie cu decoratori

- Comparatie intre *map()* si colectia iterativa de tip *list*:

```
import time, sys
force = list if sys.version_info[0] == 3 else
    (lambda X: X)
class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%s: %.5f, %.5f' %
(self.func.__name__, elapsed, self.alltime))
        return result
```

```
@timer
def listcomp(N):
    return [x * 2 for x in range(N)]
@timer
def mapcall(N):
    return force(map((lambda x: x * 2),
    range(N)))

result = listcomp(5)
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
# Etc.
```

Masurarea...

- Cu decorator cu argumente:

```
import time
def timer(label='', trace=True): # La decorare se retin argumentele
    class Timer:
        def __init__(self, func): # La decorare se retine functia decorata
            self.func = func
            self.alltime = 0
        def __call__(self, *args, **kargs): # Se executa la apelul functiei decorate
            start = time.clock()
            result = self.func(*args, **kargs)
            elapsed = time.clock() - start
            self.alltime += elapsed
            if trace:
                format = '%s %s: %.5f, %.5f'
                values = (label, self.func.__name__, elapsed, self.alltime)
                print(format % values)
            return result
    return Timer
```

Sumar

- Cod asociat cu accesarea atributelor
- Proprietati
- Descriptori
- *__getattr__* si *__getattribute__*
- Validarea atributelor
- Definitia decoratorilor
- Decorarea functiilor
- ***Decorarea claselor***

Clase ***singleton***

- Sunt clase care au o singura instanta:

```
instances = {} # Dictionar de instante

def singleton(aClass): # Decorator!
    def onCall(*args, **kwargs): # Apelata la crearea instantei
        if aClass not in instances: # O instanta per clasa, SINGLETON
            instances[aClass] = aClass(*args, **kwargs)
        return instances[aClass]
    return onCall

@singleton # Spam = singleton(Spam)
class Spam: # onCall retine Spam in argumentul aClass
    def __init__(self, val):
        self.attr = val

@singleton # Person = singleton(Person)
class Person: # onCall retine Person in argumentul aClass
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate
```

Clase...

- Alternative de *singleton*:

```
def singleton(aClass): # Cu nonlocal
    instance = None
    def onCall(*args, **kwargs):
        nonlocal instance # v3.x
        if instance == None:
            instance = aClass(*args, **kwargs) # Instanta unica
        return instance
    return onCall


def singleton(aClass): # Cu atribut de functie
    def onCall(*args, **kwargs):
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs) # Instanta unica
        return onCall.instance
    onCall.instance = None
    return onCall


class singleton: # Cu clasa si atribute de instanta
    def __init__(self, aClass): # La decorare
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args, **kwargs): # La crearea instantei
        if self.instance == None:
            self.instance = self.aClass(*args, **kwargs) # Instanta unica
        return self.instance
```

Clase...

- E momentul sa ne oprim !
- Cititi si despre *metaclase*... candva...

Va urez bafta in sesiune si mai ales multa sanatate !!