Managed Attributes
How to silently extend classes

Walter Cazzola

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

Class Extensions through Managed Attributes
Case Study: Account

Let us consider the classic implementation for the account class

```
class account:
  def __init__(self, initial_amount):
     self.amount = initial_amount
  def balance(self):
     return self.amount
  def withdraw(self, amount):
     self.amount -= amount
  def deposit(self, amount):
     self.amount += amount

if __name__ == "__main__":
  a = account(1000)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(3000)
  print("The current balance is {0}".format(a.balance()))
```

```
[23:15]cazzola@ulik:~/esercizi-pa/managed>python3 account.py
The current balance is 1000
The current balance is 1650
The current balance is -1350
```

What's about extending the naïve implementation without interfering with its basics?

- key concept: separation of concerns

Class Extensions through Managed Attributes
Inserting Code to Run on Attribute Access

An optimal solution should allow you to run code automatically on attribute access.

Three Approaches

- properties
- descriptor protocol (deja vu)
- operator overloading

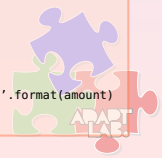
Class Extensions through Managed Attributes
Properties: To Avoid Red Balances

```
import account

class safe_account(account.account):
  def __init__(self, initial_amount):
    self._amount = initial_amount
  def save_get(self):
    return self._amount
  def save_set(self, amount):
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
    self._amount=amount
  amount = property(save_get, save_set, None, "Managed balance against excessive withdrawals")

if __name__ == "__main__":
  a = safe_account(1000)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(3000)
  print("The current balance is 0".format(a.balance()))
```

```
[23:31]cazzola@ulik:~/esercizi-pa/managed>python3 account+property.py
The current balance is 1000
The current balance is 1650
Traceback (most recent call last):
  File "account+property.py", line 19, in <module>
    a.withdraw(3000)
  File "/home/cazzola/esercizi-pa/managed/account.py", line 7, in withdraw
    self.amount -= amount
  File "account+property.py", line 9, in save_set
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
AssertionError: Not admitted operation: the final balance (-1350) MUST be positive
```



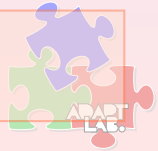
Class Extensions through Managed Attributes

Properties: To Dynamically Calculate the Balance


```
class account_with_calculated_balance:
  def __init__(self, initial_amount):
    self._deposits = initial_amount
    self._withdrawals = 0
  def deposit(self, amount):
    self._deposits += amount
  def withdraw(self, amount):
    self._withdrawals += amount
  def calculated_balance(self):
    return self._deposits-self._withdrawals
  def zeroing_balance(self):
    self._deposits = 0
    self._withdrawals = 0
  balance = property(calculated_balance, None, zeroing_balance, "Calculate Balance")

if __name__ == "__main__":
  a = account_with_calculated_balance(1000)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(3000)
  print("The current balance is {0}".format(a.balance))
  del a.balance
  print("The current balance is {0}".format(a.balance))
```

```
[23:57]cazzola@ulik:~/esercizi-pa/managed>python3 account+property2.py
The current balance is 1000
The current balance is 1650
The current balance is -1350
The current balance is 0
```



Class Extensions through Managed Attributes

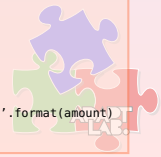
Descriptor Protocol: To Avoid Red Balances


```
import account
class safe_descriptor:
  """Managed balance against excessive withdrawals"""
  def __get__(self, instance, owner):
    return instance._amount
  def __set__(self, instance, amount):
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
    instance._amount=amount

class safe_account(account.account):
  def __init__(self, initial_amount):
    self._amount = initial_amount
  amount = safe_descriptor()

if __name__ == "__main__":
  a = safe_account(1000)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(3000)
  print("The current balance is 0".format(a.balance()))
```

```
[23:59]cazzola@ulik:~/esercizi-pa/managed>python3 account+descriptors.py
The current balance is 1000
The current balance is 1650
Traceback (most recent call last):
  File "account+descriptors.py", line 22, in <module>
    a.withdraw(3000)
  File "/home/cazzola/esercizi-pa/managed/account.py", line 7, in withdraw
    self.amount -= amount
  File "account+descriptors.py", line 8, in __set__
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
AssertionError: Not admitted operation: the final balance (-1350) MUST be positive
```



Class Extensions through Managed Attributes

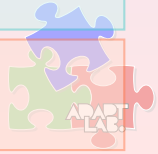
Descriptor Protocol: To Dynamically Calculate the Balance


```
class balance_descriptor:
  """Calculate Balance"""
  def __get__(self, instance, owner):
    return instance._deposits-instance._withdrawals
  def __delete__(self, instance):
    instance._deposits = 0
    instance._withdrawals = 0

class account_with_calculated_balance:
  def __init__(self, initial_amount):
    self._deposits = initial_amount
    self._withdrawals = 0
  def deposit(self, amount):
    self._deposits += amount
  def withdraw(self, amount):
    self._withdrawals += amount
  balance = balance_descriptor()

if __name__ == "__main__":
  a = account_with_calculated_balance(1000)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(3000)
  print("The current balance is {0}".format(a.balance))
  del a.balance
  print("The current balance is {0}".format(a.balance))
```

```
[0:05]cazzola@ulik:~/esercizi-pa/managed>python3 account+descriptors2.py
The current balance is 1000
The current balance is 1650
The current balance is -1350
The current balance is 0
```



Class Extensions through Managed Attributes

Operator Overloading Protocol

- `__getattr__` is run for fetches on undefined attributes.
- `__getattribute__` is run for fetches on every attribute, so when using it you must be cautious to avoid recursive loops by passing attribute accesses to a superclass.
- `__setattr__` try to guess
- `__delattr__` is run for deletion on every attribute

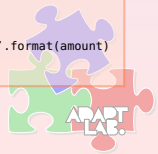
Class Extensions through Managed Attributes
Operator Overloading Protocol: To Avoid Red Balances


```
import account

class safe_account(account.account):
  def __setattr__(self, attribute, amount):
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
    self.__dict__[attribute] = amount

if __name__ == "__main__":
  a = safe_account(1000)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance()))
  a.withdraw(3000)
  print("The current balance is 0".format(a.balance()))
```

```
[0:29]cazzola@ulik:~/esercizi-pa/managed>python3 account+overloading.py
The current balance is 1000
The current balance is 1650
Traceback (most recent call last):
  File "account+overloading.py", line 16, in <module>
    a.withdraw(3000)
  File "/home/cazzola/esercizi-pa/managed/account.py", line 7, in withdraw
    self.amount -= amount
  File "account+overloading.py", line 7, in __setattr__
    assert amount > 0, 'Not admitted operation: the final balance ({0}) MUST be positive'.format(amount)
AssertionError: Not admitted operation: the final balance (-1350) MUST be positive
```

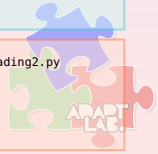


Class Extensions through Managed Attributes
Operator Overloading: To Dynamically Calculate the Balance


```
class account_with_calculated_balance:
  def __init__(self, initial_amount):
    self._deposits = initial_amount
    self._withdrawals = 0
  def deposit(self, amount):
    self._deposits += amount
  def withdraw(self, amount):
    self._withdrawals += amount
  def __getattr__(self, attribute):
    if attribute == 'balance':
      return self._deposits-self._withdrawals
    else: raise AttributeError(attr)
  def __delattr__(self, attribute):
    if attribute == 'balance':
      self._deposits = 0
      self._withdrawals = 0
    else: raise AttributeError(attr)

if __name__ == "__main__":
  a = account_with_calculated_balance(1000)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(100)
  a.deposit(750)
  print("The current balance is {0}".format(a.balance))
  a.withdraw(3000)
  print("The current balance is {0}".format(a.balance))
  del a.balance
  print("The current balance is {0}".format(a.balance))
```

```
[0:38]cazzola@ulik:~/aux_work/projects/python/esercizi-pa/managed>python3 account+overloading2.py
The current balance is 1000
The current balance is 1650
The current balance is -1350
The current balance is 0
```



Class Extensions through Managed Attributes
__getattr__ vs __getattribute__


```
class GetAttr:
  attr1 = 1
  def __init__(self):
    self.attr2 = 2
  def __getattr__(self, attr):
    print('get: ' + attr)
    return 3
```

```
[0:51]cazzola@ulik:~/esercizi-pa/managed>python3
>>> from GetAttr import GetAttr
>>> X=GetAttr()
>>> print(X.attr1)
1
>>> print(X.attr2)
2
>>> print(X.attr3)
get: attr3
3
```

```
class GetAttribute(object):
  attr1 = 1
  def __init__(self):
    self.attr2 = 2
  def __getattribute__(self, attr):
    print('get: ' + attr)
    if attr == 'attr3':
      return 3
    else:
      return object.__getattribute__(self, attr)
```

```
[0:58]cazzola@ulik:~/esercizi-pa/managed>python3
>>> from GetAttribute import GetAttribute
>>> X = GetAttribute()
>>> print(X.attr1)
get: attr1
1
>>> print(X.attr2)
get: attr2
2
>>> print(X.attr3)
get: attr3
3
```

References

- Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson.
  Practical Programming: An Introduction to Computer Science Using Python.
  The Pragmatic Bookshelf, second edition, 2009.
- Mark Lutz.
  Learning Python.
  O'Reilly, fourth edition, November 2009.
- Mark Pilgrim.
  Dive into Python 3.
  Apress*, 2009.
- Mark Summerfield.
  Programming in Python 3: A Complete Introduction to the Python Language.
  Addison-Wesley, October 2009.