Още ООП. Магически методи в Python

Филип Андонов

16 юни 2022 г.

Съдържание

- Пренаписване на методи
- Функцията super
- Протоколи
- Свойства
- Итератори
- Генератори

Пренаписване на методи

При обектно-ориентираното програмиране често се случва базовият клас да дефинира метод, който не върши работа в някой от производните класове. Причината може да е, че методът в базовия клас е прекалено специфичен, или пък обратно, че е прекалено общ и трябва да бъде допълнен. Тогава производният клас трябва да има възможност да подмени съдържанието на даден метод със специфично за него. Това се извършва с т. нар. method override (пренаписване на метод).

Пренаписване на методи

Inherited не съдържа собствен метод hello, поради което при обръщение към него в екземпляр на наследения клас се извиква кода, дефиниран в базовия.

Listing 1: Наследяване на метод

```
1  class BaseClass(object):
2      def hello(self):
3          print("Hello, I'm BaseClass.")
4
5  class Inherited(BaseClass):
6      pass
7
8  a=BaseClass()
9  b=Inherited()
10 a.hello()
11 b.hello()
```

Пренаписване на методи

Двете извиквания на hello на практика изпълняват различни методи.

Listing 2: Пренаписване на метод

```
1 class BaseClass(object):
2     def hello(self):
3         print("Hello, I'm BaseClass.")
4
5 class Inherited(BaseClass):
6     def hello(self):
7         print("Hello, I'm Inherited.")
8 a.hello()
9 b.hello()
```

Пренаписване на методи

Атрибутите на обекта в Python се дефинират в конструктора. Следователно, ако той бъде пренаписан, всички атрибути, дефинирани в базовия конструктор, но отсъстващи в производния, ще престанат да съществуват за наследения. В примера по-долу наследеният клас пренаписва конструктора на базовия и в резултат създаването на атрибута workhours не е извършено за класа Manager и затова ред 8 ще породи изключение.

Listing 3: Проблем при пренаписване на конструктор

```
1  class Employee(object):
2      def __init__(self):
3          self.workhours = 8
4      def work(self):
5          print("I work", self.workhours," a day")
6
7  class Manager(Employee):
8      def __init__(self):
9              print("Go to work!")
10 a = Employee()
```

Функцията super

Функцията super се извиква с аргументи текущия клас и екземпляр, и всеки метод, който извикаме на върнатия обект ще бъде взет от родителския клас, а не от текущия. Това ни дава възможност да решим проблема с пренаписването на конструктора. Можем да извикаме super(Manager, self). Така конструкторът __init__ на производния клас ще извика този на базовия, който ще създаде необходимите атрибути. Има два начина, по който да използваме super. Наследеният от Python 2 начин е:

super(subClass, instance).method(args)

Новият и по-удобен синтаксис е:

super().methoName(args)

Функцията super

Трябва да имаме предвид обаче, че когато става въпрос за множествено наследяване, то името на функцията е малко подвеждащо. Това е така, защото в действителност тя взима от първия обект в списъка на Method Resolution Order алгоритъма. Така може да се окаже, че наследяваме от обект „брат“, а не „родител“.

Функцията super

Listing 4: Пренаписване на конструктор със self

```
1  class Employee(object):
2      def __init__(self):
3          self.workhours = 8
4
5      def work(self):
6          print("I work for ",self.workhours," a day"
               )
7
8  class Manager(Employee):
9      def __init__(self):
10         super().__init__()
11
12 a = Employee()
13 b = Manager()
14 a.work()
```

Протоколи

В Python няма интерфейси, но подобна функционалност се постига с протоколи. Полиморфизмът в Python е базиран на поведение, а не на наследяемост или реализиране на интерфейс. Поведението се състои в това да бъдат реализирани дадени методи и какво те да извършват. В резултат това, което в други езици задължава даден обект да принадлежи на даден клас или да реализира даден интерфейс, в Python се случва просто като се следва даден протокол.

Протоколи

За да бъде даде обект редица, то той просто трябва да следва протокола. Основният протокол за sequence и mapping е следния:

- len
- getitem
- setitem
- delitem

`__len__`

Този метод трябва да връща броя на елементите в колекцията. За редица това е просто броят на елементите, за съответствие – броят на двойките ключ-стойност. Ако `__len__` върне 0, обектът се третира като False в булев смисъл.

`__getitem__(self, key)`

Този метод трябва да връща стойността, съответстваща с ключа key. За редица, ключът трябва да е целочислена стойност от 0 до n-1 или отрицателна, където n е дължината на редицата. За съответствие можем да имаме всякакъв вид ключ.

`__setitem__(self, key, value)`
Този метод трябва да записва стойността value, асоциирана с ключа key, така че по-късно да може да бъде извлечена с `__getitem__`. Този метод се реализира, само ако обектът ни е изменим.

`__delitem__(self, key)`

Този метод трябва да се извиква, когато някой извиква израза `del` като част от обекта и да изтрива асоциирания с подадения ключ елемент. Методът се реализира, само ако обектът ни е изменим.

Допълнителни изисквания

За редица, ако ключът е отрицателно число, трябва да брои от края (x[-n]=x[len(x)-n]). Ако ключът е от грешен тип, да се поражда изключение TypeError. Ако индексът на редица е от правилен тип, но извън обхвата, да се породи изключение IndexError.

За илюстрация ще създадем клас, който представя точно три имена на човек и се държи като редица. Реализацията не е елегантна, но се вижда реализацията на всяко от изискванията на протокола.

Listing 5: Протокол за редица

```
1  class Fullname(object):
2      def __len__(self):
3          return 3
4      def __getitem__(self, key):
5          if key >= 0:
6              if key == 0:
7                  return self.fname
8              elif key == 1:
9                  return self.mname
10             elif key == 2:
11                 return self.lname
12             else:
13                 raise IndexError
```

Listing 6: Протокол за редица

```
14          else:
15              if key == -3:
16                  return self.fname
17              elif key == -2:
18                  return self.mname
19              elif key == -1:
20                  return self.lname
21              else:
22                      raise IndexError
```

Listing 7: Протокол за редица

```
23      def __setitem__(self, key, value):
24          if key >= 0:
25              if key == 0:
26                  self.fname = value
27              elif key == 1:
28                  self.mname = value
29              elif key == 2:
30                  self.lname = value
31              else:
32                  raise IndexError
```

Listing 8: Протокол за редица

```
33          else:
34              if key == -3:
35                  self.fname = value
36              elif key == -2:
37                  self.mname = value
38              elif key == -1:
39                  self.lname = value
40              else:
41                  raise IndexError
```

Listing 9: Протокол за редица

```
42      def __init__(self,f,m,l):
43          self.fname = f
44          self.mname = m
45          self.lname = l

47  name=Fullname("John", "Fitzgerald", "Kennedy")
48  print(name[-1])
49  for n in name:
50      print(n)
```

Примерният клас Fullname реализира цялата функционалност на класа от нулата. Ако искаме да реализираме клас с функционалност, близка до някои от вградените типове, е по-уместно да ги наследим. Трябва просто да наследим класовете от стандартната библиотека list, string, dict. Нека направим списък, който брои колко пъти е осъществяван достъп до елементите му. За целта ще пренапишем конструктора и метода __getitem__. Виждаме, че в началото стойността на брояча е 0, но след двете обръщения към елементи на ред 21 тя вече е 2.

Listing 10: Наследяване на вграден тип

```
1  class CounterList(list):
2      def __init__(self, *args):
3          super(CounterList, self).__init__(*args
               )
4          self.counter = 0
5
6      def __getitem__(self, index):
7          self.counter += 1
8          return super(CounterList, self).
               __getitem__(index)
9  >>> cl = CounterList(range(10))
```

Listing 11: Наследяване на вграден тип

```
>>> cl
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3, 2, 1, 0]
>>> cl.counter
0
>>> cl[4]+cl[2]
9
>>> cl.counter
2
```

Свойства

В класическия стил на обектно-ориентираното програмиране всеки атрибут се чете и променя не като променлива, а чрез специални методи, наречени съответно getter и setter, които предпазват атрибута на обекта от жестокия свят навън. Това води до утежняване на обръщенията към атрибутите на обектите. Python предлага механизъм, наречен свойства, който обединява добрите страни на двата подхода. След като бъдат създаден атрибут и методи за четенето и промяната му, всичко това може да се обвърже със свойство, обръщението към което е като към атрибут, но на практика извиква лежащите отдолу getter и setter методи.

Listing 12: Свойства

```
1  class car(object):
2      def __init__(self,b,m):
3          self.brand = b
4          self.model = m
5
6      def getBrand(self):
7          return self.brand
8
9      def setBrand(self,b):
10         self.brand = b
11
12     def setModel(self,m):
13         self.model = m
```

Listing 13: Свойства

```
14      def getModel(self):
15          return self.model

17      def delBrand(self):
18          self.brand = None
19          self.model = None

21      Model = property(getModel,setModel)
22      Brand = property(getBrand,setBrand, delBrand)
```

Listing 14: Свойства

```
23 fe = car('Ford','Escort')
24 print(fe.brand,fe.model)
25 fe.Model = 'Fiesta'
26 print(fe.Brand,fe.Model)
27 del fe.Brand
```

Свойства

Функцията property приема като аргументи първо getter, после setter метода и връща метод, който се обвързва с дефинирано име на свойството. Функцията property може да бъде извикана с нула, един, два, три или четири аргумента.

- Без аргументи: свойството не е нито читаемо нито записваемо.
- С един аргумент (само getter) – само за четене
- Параметър за установяване на стойност (setter)
- Третият незадължителен параметър е за изтриване на атрибут (не приема аргументи).
- Четвъртият незадължителен параметър е документиращ символен низ.

Имената на параметрите са: fget, fset, fdel и doc— можем да ги използваме като аргументи с ключова дума.

Протоколи за итериране

Итерирането означава многократно повторение. Досега итерирахме само върху редици и речници с цикли for, но всъщност можем да итерираме и върху други обекти. Достатъчно е те да имплементират метод __iter__. Той връща итератор, който пък е всеки обект с метод с име __next__, който се извиква без аргументи. Този метод трябва да връща „следващата стойност“, каквото и да означава това в конкретния случай. Ако итераторът няма повече стойности, методът трябва да породи StopIteration exception. Нека реализираме клас, представящ функцията факториел. Всеки път когато извикаме next() или извършим итериране върху обект от този клас, ние ще получаваме факториел от следващото естествено число.

Протоколи за итериране

Listing 15: Факториел с протокол за итериране

```
1  class Factoriel(object):
2      def __init__(self):
3          self.n = 0
4          self.f = 1
5
6      def __next__(self):
7          self.n = self.n+1
8          self.f = self.n*self.f
9          return self.f
10
11     def __iter__(self):
12         return self
```

Listing 16: Факториел с протокол за итериране

```
13 fact = Factoriel()
14 for f in fact:
15     print(f)
16     if f > 10000:
17         break
```

Когато функция завърши действието си, локалните променливи се унищожават. Ако направим аналогия с MP3 плеър, това е натискане на бутона СТОП – мястото, до което е стигнала текущата песен се забравя и при последващо пускане се започва отначало. Но ако вместо СТОП натиснем ПАУЗА, тогава изпълнението се спира, но стартирането започва от мястото на спиране, а не от начало. При функциите това означава да не унищожаваме локалните променливи. Тогава можем да продължим изпълнението на функцията от последното място, на което сме я спрели.

Генератори

В Python това се постига с генератори, които са вид итератори. Генератор е функция, която съдържа ключовата дума yield. Когато се извика, кодът във функцията не се изпълнява, а вместо това се връща итератор. Всеки път, когато се изиска стойност, кодът в генератора се изпълнява докато не се срещне yield или return. Yield означава, че стойността трябва да се породи, т.е. да стартира функцията от мястото на спиране. Return означава, че генераторът трябва да спре, без да поражда нищо повече. Return клаузата не може да има аргументи, когато е в генератор. Генераторите се състоят от две компоненти - функция-генератор – дефинирана с израз def и съдържаща yield и генератор-итератор – върнатият от функцията резултат. За да демонстрираме удобството на генераторите ще сравним класическата имплементация на редицата на Фибоначи с функция, която генерира стойностите от нея.

Listing 17: Фибоначи, класическа реализация

```
1  def Fibonaci(n):
2      a = 0
3      b = 1
4      tmp = 0
5      for i in range(1,n):
6          tmp = a + b
7          a = b
8          b = tmp
9      return b

11 for i in range(1,10):
12     print(Fibonaci(i))
```

Генератори

Трябва да обърнем внимание, че в следващата реализация печатането с цикъл, започващо на ред 13 и печатането, започващо на ред 19 създават различни генератори, поради което второто печатане започва стойностите отначало.

Listing 18: Фибоначи, реализация с генератори

```
1  def Fibonaci():
2      a = 0
3      b = 1
4      tmp = 0
5      yield a
6      yield b
7      while 1:
8          tmp = a + b
9          yield tmp
10         a = b
11         b = tmp
```

Listing 19: Фибоначи, реализация с генератори

```
12 for i in Fibonaci():
13     print(i)
14     if i>100:
15         break

17 fib = Fibonaci()
18 print(fib.__next__())
19 print(fib.__next__())
20 print(fib.__next__())
```

Пример 1

Нека искаме да създадем програма, която чете символен низ от клавиатурата символ по символ и го извежда наобратно. Това може да се реализира рекурсивно с много проста функция, състояща се само от три (съществени) реда – прочитане на символ, рекурсивно извикване на функцията и извеждане на прочетения символ на екрана. Построена по този начин, тя използва програмния стек (последен влязъл първи излязъл), за съхранението на символите. За да сработи целият механизъм, ключовото е извършването на действие след рекурсивното извикване. Обикновено при рекурсивни функции то е последният ѝ оператор, но ако след него има други оператори, те се различават качествено от тези, намиращи се преди рекурсивното извикване. Действията преди рекурсивното извикване се извършват при потъване в рекурсията, а тези след – при изплуване.

Пример 2

Floodfill е алгоритъм, който определя зона, свързана с даден възел в многомерен масив. Използва се в инструмента „кофа“ в програмите за рисуване за определяне кои части от bitmap да запълни с даден цвят.

Floodfill (node, target-col, replace-col):
 If цветът на възела не е равен на целевия, return.
 Установи цвета на възела на цвета цвета заместител.
 Извърши Floodfill (стъпка наляво, target-col, replace-col).
 Извърши Floodfill (стъпка надясно, target-col, replace-col).
 Извърши Floodfill (стъпка нагоре, target-col, replace-col).
 Извърши Floodfill (стъпка надолу, target-col, replace-col).
 Return.

Пример 3 - Търсене с връщане

Едно от интересните приложения на рекурсията са т. нар. търсене с връщане. Използват се за намиране на решенията на специфични проблеми, които не следват фиксирано правило за пресмятане, а се самонастройват чрез проби и грешки. Общата схема се състои в разбиване на процеса на пробите и грешките на отделни задачи. Тези задачи най-често се изразяват естествено чрез рекурсия и се състоят в изследване на краен брой подзадачи. Целият процес представлява пробване или претърсване, при който постепенно се построява и сканира дървото на подзадачите. Дървото в повечето задачи расте много бързо, обикновено експоненциално. Често дървото може да се окастри, като се използват евристични похвати. Класически задачи, използващи търсене с връщане са шахматните „разходката на коня“ и „осемте царици“, но ние ще направим нещо още по-интересно: намиране на път в лабиринт. Нека преди това обаче видим общия вид на процедурата. Обърнете внимание, че тя наистина не е за конкретна задача.

Пример 3 - Търсене с връщане

```
Procedure следващ_ход
Инициализация на възможните ходове
Do Избор на следващ ход от кандидатите
   if е приемлив {
      Регистрация на хода
      if не сме достигнали крайно решение
         следващ\_ход
      if неуспех изтриване на предишната регистрация
   }
While ходът е неуспешен и има още кандидати
```

Пример 4 - Намиране на път в лабиринт

Алгоритъмът за запълване на област в изображение (floodfill) прилича на схемата за връщане стъпка назад с изключение на това, че той не прави нищо при изплуване, т.е. не маркира хода като „грешен“. Така че намирането на път в лабиринт е модификация на floodfill алгоритъма. Първо проверяваме дали цветът на текущата клетка е този, който трябва да заменим. Ако това не е така, не правим нищо. Ако е, веднага го подменяме с новия цвят. След това извикваме същата процедура за съседите отляво, отдясно, отгоре и отдолу. Запълването на област може да се използва в лабиринт, за да се провери дали има път между две клетки. За да намерим един път (без условие за оптималност), трябва да добавим към току-що описаните действия връщането назад. Оцветяването на текущия възел с цвета-заместител е маркиране като потенциално водещ към целта. След рекурсивните извиквания ще добавим подобно действие, което маркира текущия възел като „неуспешен“, т.е. не водещ към целта.

Пример 4 - Намиране на път в лабиринт

Именно тук е разковничето – при потъване (преди рекурсивните извиквания) ние маркираме всеки посетен необходен възел като потенциално добър. На изплуване го маркираме като „лош“. Така когато стигнем до целта имаме очертан от „потенциално добри“ елементи път, както и необходени, а също и доказано не водещи към целта елементи. В реализацията ни лабиринтът е двумерен списък, в който с 1 означаваме стени, с 0 необходен елемент, със 7 е означена целта, с 3 означаваме обходен и потенциално добър елемент, а с 5 – обходен в права и обратна посока и доказано не водещ до целта. При печат на лабиринта използваме символи вместо числа за по-голяма прегледност.

Задача 1

Да се създаде клас Fibs за редицата на Фибоначи, реализиращ протокола за итериране. Следният код трябва да може да се изпълнява върху него.

```
fibs = Fibs()
  for f in fibs:
    if f > 1000:
      print(f)
      break
```

Задача 2

За успешното решаване на задачата е необходимо да са разбрани някои специфични аспекти на рекурсията – дънна рекурсия, многоклонова рекурсия и алгоритми с отстъпване, които са илюрстрирани с примерите по-горе. Проектът се състои в реализирането на итеративна версия на алгоритъма за обхождане на лабиринт. Вместо програмен стек може да се използва списък.

Thanks for using Focus!

References